AD-A080 395 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 5/2
THE DESIGN AND IMPLEMENTATION OF A PEDAGOGICAL RELATIONAL DATAB--ETC(U)
DEC 79 M A ROTH
UNCLASSIFIED AFIT/GCS/EE/79-14 NL

1 OF 2
AD A080 395

AFIT/GCS/EE/79-14

(6) THE DESIGN AND IMPLEMENTATION OF
A PEDAGOGICAL RELATIONAL DATABASE SYSTEM.

(9) [illegible] THESIS,

(12) 153

(14) AFIT/GCS/EE/79-14

(10) Mark A./Roth
2LT USAF

(11) 1 [illegible]

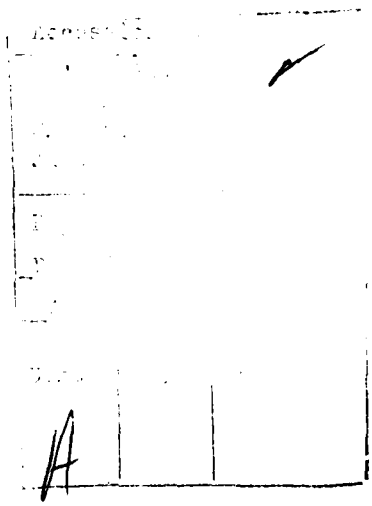
Approved for public release; distribution unlimited

AFIT/GCS/EE/79-14

THE DESIGN AND IMPLEMENTATION OF A PEDAGOGICAL RELATIONAL DATABASE SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University (ATC)
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science



by

Mark A. Roth
2Lt USAF

Graduate Computer Systems

15 December 1979

PREFACE

The need for a good pedagogical tool in the area of database systems has existed for some time. Although commercial database management systems could be used, their complexity and lack of flexibility precludes their use as a good instructional tool. Thus, Dr. Thomas Hartrum, on the faculty of the AFIT/EN Electrical Engineering department, proposed that such a tool be developed as a master's degree thesis. I undertook this project with the goal of handing him a fully operational system at termination. However, since a relational system was being designed to provide the necessary flexibility, the problem of efficiency in relational systems became a high priority. Thus we decided that the best approach would be to explore the efficiency problem, and present only the front end of the system as my final product. Hopefully, through well-structured design, coding, and documentation, the system can be easily finished.

Many thanks are due to the members of my thesis committee for their helpful comments both during the development and in the writing of this document. They were advisor, Dr. Thomas Hartrum, and committee members, Dr. Alan Ross, Dr. James Rutledge, and Dr. Kenneth Melendez. Thanks are also due to Capt Brian Boesch for his many hours of help with the computer system I used. He is largely responsible for the material in Appendix A. In this vein, thanks are also due the AFIT/ENE technicians: Robert Durham, Dan Zombon, Orville Wright, and Richard Wager. Lastly, I would like to thank 2Lt Pete Raeth for implementing the code in the RUN module.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

A relational database system was designed with the goal of obtaining as near optimal behavior from the system as possible. In addition, the database was to be implemented as a general purpose system but with specific provisions for its use as a pedagogical tool for teaching database management and manipulation.

Toward these goals, investigations were made into previous theoretical studies in the literature. The advantages and disadvantages of relational systems were explored, and based on a certain criteria a relational algebra was chosen as the basis for the data manipulation and definition language. Solutions to the problems of relational databases, including integrity, redundancy, and efficiency, were presented in this context.

With this background, a top-down structured design of the system was completed. Techniques used to manage data entry; i.e., the user interface, and techniques to transform those inputs in order to optimize their execution were developed and implemented. These transformations formed the basis of an automatic programmer used to analyze and efficiently refine high level query specifications supplied by the user. This approach sought to minimize query response time and space utilization by: (1) performing global query optimization, and (2) coordinating sort orders in temporary relations.

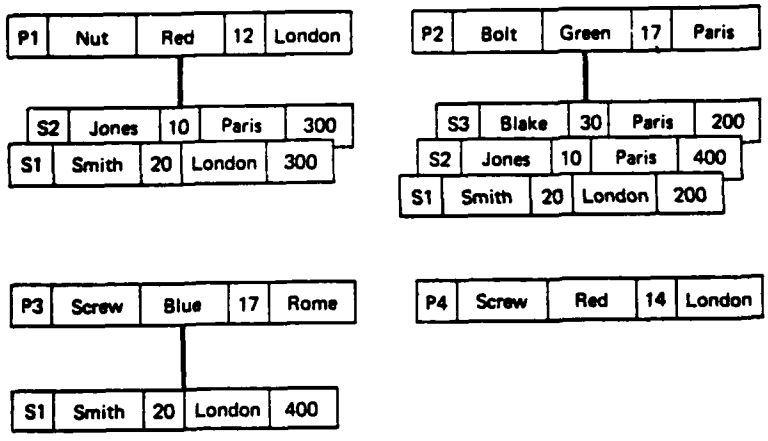
THE DESIGN AND IMPLEMENTATION OF A PEDAGOGICAL RELATIONAL DATABASE SYSTEM

I INTRODUCTION

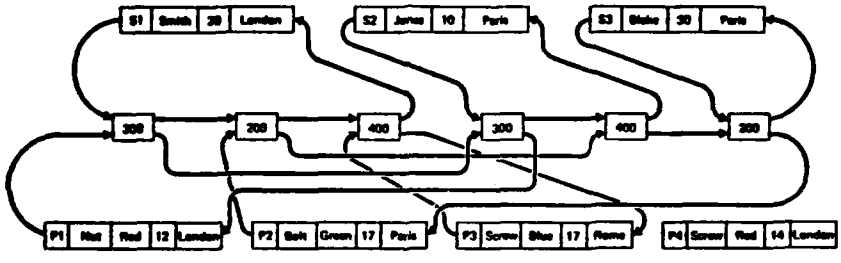
BACKGROUND

The relational model of data is a concept that was formalized by E.F. Codd (Ref 1) in 1970 and has been gradually gaining acceptance in the research community. Prior to 1970, database implementations were generally based on one of two models, the hierarchical data model and the network data model. An example of the hierarchical model appears in Figure 1(a). In this example, parts records consisting of a part number, name, color, weight and city can have, subordinate to them, zero or more supplier records consisting of a supplier number, name, status, city and quantity of the part being supplied. One of the major drawbacks of this model of the data is the inherent redundancy of information. Note in the example how the same information for suppliers is repeated many times throughout the hierarchy. In addition similar queries about the data cannot necessarily be formulated in a similar manner. In the example it would be easy to find all suppliers of a given part by doing a simple lookup of that part, but finding all the parts supplied by a particular supplier forces a scan of the entire hierarchy, searching for all occurrences of the particular supplier.

A solution to some of the problems of the hierarchical data model was found in the network data model. The network model uses sets and link constructs to indicate relationships among the data. The example data as it would look in the network model is shown in Figure 1(b).



(a) Sample data in hierarchical form (parts superior to suppliers)



(b) Sample data in network form.

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S2	P1	300
S2	P2	400
S3	P2	200

(c) Sample data in relational form.

Figure 1. Sample data depicted with three models (Ref 4).

Here queries are answered by searching through the paths (indicated by arrows) until the proper information has been reached. Thus, although the problems of the hierarchical model no longer prevail, the complexity of searching through the network causes other problems. One of the biggest ones is the lack of data independence. In order to be able to use the network model, the user must know the structure of the various networks he must work with, and write his application programs accordingly. If there is some need to change the database, either by deleting or by adding new data or relationships, the user may have to rewrite his programs to match the new model of the data.

Thus when Codd introduced the relational model, the user community applauded it for its simplicity, completeness, and ability to provide excellent data independence. The example data in relational form is shown in Figure 1(c). The three relations PART, SUPPLIER, and SP respectively denote a relation of parts records (or tuples), a relation of supplier records, and a relation of supplier-part records indicating which suppliers supply which parts in what quantities.

Although university implementations such as PRTV and MacAIMS (Ref 2) are becoming commonplace, and even a few commercial implementations have appeared, relational systems in general have been criticized because of their comparative slowness and inefficiency when actually implemented. Thus, new research was stimulated with the purpose of eliminating or at least mitigating the effects of those inefficiencies.

Research has concentrated in both the hardware and software design areas. Hardware improvements are concerned with reducing the time and space needs of the typically software-laden database management system which, being large in size and complex in structure, has been overtaxing

the host hardware, and overshadowing the host operating system. Most of the effort in this area has been made with large to very large databases in mind, which is, of course, where the software systems have been straining the most (Ref 3:414).

In contrast, software research has focused on all facets of relational database systems, where new algorithms and structures are being developed to increase efficiency for large as well as small systems. Some of this research is being done in conjunction with the design of the database computers, but most of the work is being done in improving the systems that run on current hardware configurations. The growing need for simple and efficient database systems to run on the proliferation of mini-, micro-, and mainframe computers in the business and government arena will demand that researchers continue to design and improve systems to run on these architectures.

There is also a great need for database systems in the educational environment. The vast majority of courses in the database area offer a basic textbook, "this is how it works in theory", approach, with little, if any, hands-on experience afforded the future database manager or user. Although there are many existing database systems which could be used to teach database manipulation with a fair degree of success, the facilities for allowing students to experience the role of database administrator are lacking.

STATEMENT OF PROBLEM

Thus, the purpose of this thesis is to solve two problems. First, a relational database system is to be designed using the best known methods as well as any new methods to exact as near possible optimum

behavior from the system. Second, the database is to be implemented as a general purpose system but with specific provisions for its use as a pedagogical tool for teaching database management and manipulation.

SCOPE

The original scope of this thesis was to design and implement a complete stand-alone database system, including a comprehensive user interface, optimization procedures, and a complete physical storage manager. After the design phase was completed, it became apparent that the implementation phase would exceed the available time. Therefore, it was decided to leave out modules of the system which dealt with the actual processing of data in the form of relations. However, due to the top-down modular design of the system, these modules should be easy to interface to the existing system once they are designed.

GENERAL APPROACH

The first step consisted of an extensive literature search to determine what research had already been done in the area of relational databases. An effort was made to collect information from all areas pertaining to database systems: query language design, query evaluation, storage structures, retrieval algorithms, and human engineering. A list of all sources accessed is presented in the bibliography.

In order that a direction be established for the rest of the thesis, the next step involved the selection of a data manipulation and definition language (DML and DDL). A wide variety of such languages have been used in past and present implementations, and many others have been defined in theoretical investigations. However, there are basically two types of relationally-oriented languages which form the basis

for most database languages. These are those based on either a relational algebra or a relational calculus. As described in Chapter III, consideration of certain criteria resulted in the selection of a relational algebra and thus a target DML and DDL was designed with that decision in mind.

The next step integrated the ideas obtained through the literature search and the decision to use a relational algebra, in an overall system design. A top-down modular design was used to allow straightforward implementation as well as modification.

Based on the system design, modules were implemented in top-down order, with the important lines of implementation done first (Refer to Scope section). And finally, the completed modules were validated to ensure their correctness of operation under all inputs, both as stand alone modules and as an integrated system.

SEQUENCE OF PRESENTATION

The remainder of this thesis is broken into five chapters. Chapter II describes the selection of the computer upon which to implement the database system. Machine size and availability and operating system features are discussed. Chapter III discusses various theoretical developments in the design of the system. These include the advantages and disadvantages of relational databases, the selection of the DML and DDL, and a description of algorithms and heuristics available and created to eliminate or at least mollify the disadvantages of the relational systems.

Chapter IV is a description of the major decisions involved in the development of the software system, highlighting the data structures and algorithms having significant impact in this implementation. Prominent

topics in this section include system design and data and control structures used at the data entry, conceptual and system implementation levels. Chapter V describes the validation of the system and Chapter VI presents conclusions and recommendations.

II HARDWARE CONSIDERATIONS

This chapter describes the selection of the computer system which was used as the development tool for implementation of the database system. Both machine type and operating systems are discussed.

MACHINE TYPE

Because one of the primary goals of the database system was to serve in a pedagogical environment, it was necessary to choose a machine which would be freely available to students and which required little if any, knowledge of machine architecture and operation. In addition, it was desired that the system have interactive capabilities to provide the user with a more flexible and intimate relationship with the database.

Choices for an appropriate machine included a multi-access CDC/Cyber 6600 and two 8080 based machines -- the INTEL 8080 system and the ALTAIR 8800b system. The large CDC computer was eliminated as a viable choice because of its already heavy use by the Wright-Patterson community and its not uncommon habit of going down for extended periods of time. In addition, the smaller machines could be made available on a full-time basis for both development of the system and future use by student users. The decision on which 8080 system to use was essentially rendered academic by the design of a communications network allowing complete portability of data and programs between machines. The methods involved in this scheme are presented in Appendix A.

OPERATING SYSTEM

One of the ways that machine communication was simplified was through use of a common disk operating system: CP/M (Control Program for a Microcomputer) (Ref 4), and a common high level operating system: UCSD (University of California, San Diego) Pascal, Version II.0 (Ref 5). CP/M was chosen because it is an excellent vehicle for implementing Pascal on a variety of machine architectures. (See Appendix A for more details). Pascal was chosen as the implementation language for several reasons. Most important is the block structured design of the language allowing a smooth transition from a top-down design to the actual code.

Other reasons for choosing Pascal are its high degree of variable typing and its wealth of control structures. The UCSD version of Pascal is a widely used implementation among microcomputers, thus providing a more portable system. Since all machine dependent features are imbedded in CP/M, any code written with one UCSD Pascal system can be run on any other. UCSD Pascal was also chosen because, by being designed for microcomputers, it has a built-in capability for handling large programs with features such as program segmentation and separately compiled procedures and virtual memory using segment swapping. Although UCSD Pascal can run completely in 48K bytes of main memory using the above features, it turns out that the database system designed here required a minimum of 64K bytes, due to compilation constraints and execution with large files. Both random and sequential access to dual drive disks is available in UCSD Pascal. Thus, the database system may reside on one disk so that procedures may be swapped in and out of memory, and user data (relations) may reside on the other disk. By simply switching user disks, data space becomes virtually unlimited.

SUMMARY

The combination of a dedicated microprocessor system and the versitality of the UCSD Pascal operating system and language proved to be a windfall. Program design was easily modularized; modification and testing of modules was simple and quick; segmentation, input/output, and string manipulation were available, easy to learn, and easy to use. Although Pascal exists on the CDC machine, it does so only in a batch operating system. To be used properly in a pedagogical environment, a database system must be interactive, allowing the student to create, execute, and modify commands, and see the results immediately.

III THEORETICAL DEVELOPMENT

This chapter describes the work that was done prior to initial system design. Through an extensive literature search an excellent understanding of the advantages and disadvantages of relational database systems was obtained. By understanding the advantages and disadvantages of relational systems, the system design could capatilize on the advantages while seeking to eliminate the disadvantages. A discussion of this occurs in the first section of this chapter.

In order to provide some direction to the system design, investigation into a target data manipulation and definition language was necessary. This enabled direction in the system design, considering the various methods to make use of the advantages and at least mollify the disadvantages of relational systems. By defining the target DML and DDL ahead of time, a definition of what the system should do was established and, consequently, the best methods could be used and/or developed to accomplish that goal. The selection of the DML and DDL is developed in the second section of this chapter.

The problems of relational systems, and possible solutions to those problems, have been "bantered back and forth" among experts in the field ever since Codd introduced the concept. In light of the target language described in the second section and the hardware considerations of Chapter II, the experts' methods used in this thesis to solve the problems of relational systems are presented in the third section of this chapter.

ADVANTAGES AND DISADVANTAGES OF RELATIONAL SYSTEMS

This section points out the potential problems and advantages which should be matters of consideration in the design of a relational model of data. These include simplicity of the data structure description, generality of data description and manipulation, flexibility and data independence, fourth normal form and redundancy of data structure, data manipulation vs. data management, and user expections (Ref 6).

Simplicity Of Data Structure Description

The aspect of the relational model of information most appealing to the casual user is its simplicity. A single type of structure, with a simple and clear organization, suffices for the storage of data, for the system catalogs defining the (relational) data sets, for inverted lists and indexes used for more efficient data retrieval, and the data dictionary/directory. The simple data structure means there is only one simple access method to understand. Relations are physically independent of one another, facilitating relocation, segmentation, and backup.

Part of the simplicity of the relational model results from the omission of details relating to the performance of the data management system and the storage of data. There are three levels of detail in a database management system (Figure 2). The relational model addresses mainly the conceptual level, with no specifications on the data entry level or the storage representation level. While it is certainly desirable to help the application programmer avoid these representation details, other system designs have found that goal difficult to achieve (Ref 6).

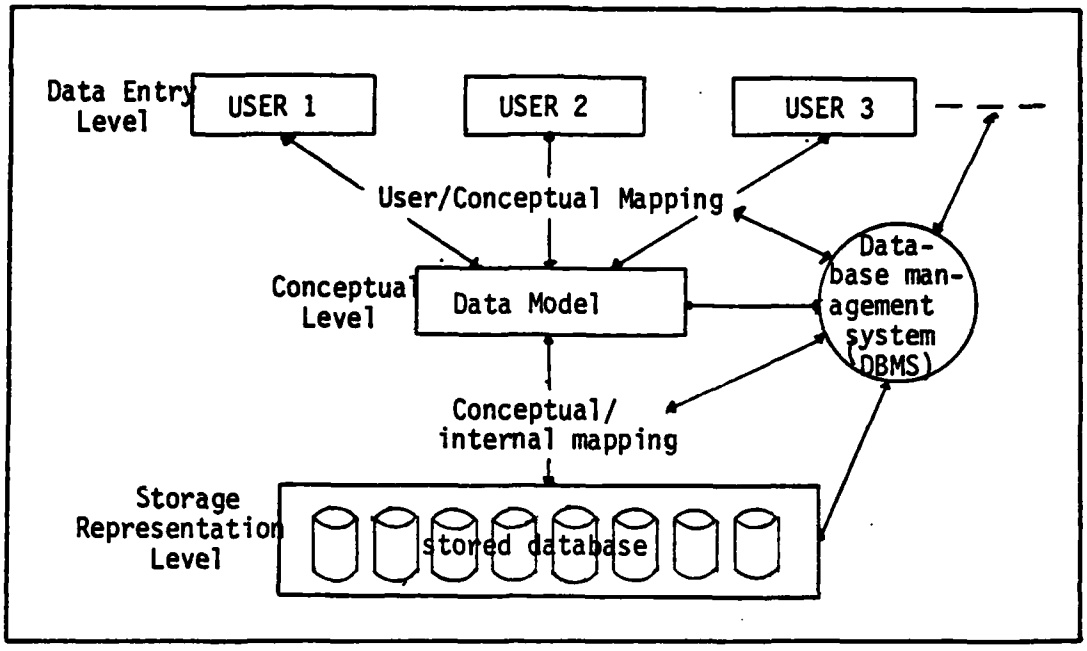


Figure 2. The three levels of a Database System

Generality of Data Description and Manipulation

The generality of the relational model is another significant advantage. To start with, the notion of Relational Completeness (Ref 1) specifies a class of queries which can be answered from the information in a given collection of relations. The class is a very general one; it includes all relations which can be represented with relational expression using as operands, the database relations. Furthermore, (Ref 7), the effort required to answer a query can frequently be estimated in advance.

A further advantage of using relations for the basic data structure is the power of the general operations which may be applied to the sets of tuples. Since each relation is a set of related items, it is sensible to have operations which perform the same action on each item in the set. This is more difficult for the complex data structures managed by other systems.

Flexibility and Data Independence

In addition to simplicity and generality, the relational model provides more flexibility than available before in other systems. Because relationships between information in tuples of different relations (or distinct tuples in the same relation) are indicated implicitly by identity of key values, adding new relationships or new data to existing relationships is made easier. The model lends itself to easy extension with new data manipulation functions, to non-procedural data manipulations, and to the use of set operations.

The separation of relationship information from the information describing entities is a further advantage for purposes of data security. We can, for example, place restrictions on the salary relationship which relates employees to the salary ranges without restricting access to the relationship between employees and their employers, assignments, and active projects.

Although indicating relationships among tuples by key values does indeed give a higher degree of independence of the relations, remembering those implicit relationships may be difficult. For example, if a network model (not to be confused with the network model of data) has a relation of edges specified by From and To node identifiers, there are implicit links from the edge relation to the node relation containing the nodes of the network. The system catalogs must maintain a record of these implicit links, which can cause problems in a large system.

Not only must the system remember relationships but it must also maintain the integrity of the relationships, for example by preventing the deletion of a tuple containing a key value which is referenced by another tuple somewhere. These checks may be exceedingly complex and difficult.

Fourth Normal Form and Redundancy of Data Structure

Reducing a relation to Fourth Normal Form (See Ref 2:153-172), while reducing the complexity of the information structure by separating information into related classes, introduces a considerable number of key value references. Instead of being grouped into the record of the owner, the elements of a repeating group are placed in another relation with a domain of key references to the owner. Usually there are many repetitions in these domains and it may be argued that this redundancy of information wastes storage space. That argument is true only if the relation is stored in the fully expanded form seen by the user, ignoring the possibility of redundancy-removing data compression techniques in the storage of the relations. Such redundancy removal techniques are advocated in the ADMINS system (Ref 8) to allow efficient use of disk storage without making the user's view of the data more complex. Expansion to fully redundant form may take place when a page of data is retrieved from the disks or when it is presented to the user or used in selection operations.

Data Manipulation vs. Data Management

The relational model of data does not really simplify the problems in a large database of storing and retrieving ten million records of 200 bytes each. These data management problems remain. A relational model may help to segment the database into portions and to reduce the physical linkages between files. But mainly, the relational systems are a new method of modeling data structures in a general and powerful way.

User Expectations

When a relational model is used for presenting data to a user, then implemented with some shortcuts such as sorting the relations on the primary key, the possibility that the user will be able to formulate two similar commands, one of which will be done instantly (find keys 3-6) and one of which will take a long time (find colors red-green), will become more probable. Thus using a more powerful user language than the implementation really supports is likely to raise expectations beyond capabilities and result in disappointments. This is especially true with the new operations such as the set operations and JOIN with which a user may not be familiar.

The question of user expectations is particularly important in the context of inquiry processing, since that application has both the greatest potential and the greatest problems. Clearly, the power of the relational systems is most evident for complex queries, particularly those involving data from more than one information file (relation). Yet, if the system maintains the tuples chained for efficient update of the relations, context sensitive searches will be slow on all domains. Correcting this is possible on any particular domain by sorting on that domain but then searches on other domains will still be slow. Furthermore, if additional operations such as JOIN need to be done on domains other that the current sort key, then the entire relation must be re-sorted, a very time-consuming operation. Indexing every relation on every domain would solve the access problem for queries, but then we have the usual problems with the updating of relations and the creation of new relations (with their indexes). Hence, with current hardware technology, systems have usually taken some implementation short-cuts for a relational model.

SELECTION OF THE DATA MANIPULATION AND DEFINITION LANGUAGE

This section examines the two types of relationally oriented database languages, the relational algebra and the relational calculus. First a short definition of each language type is presented. Then the two language types are compared, especially with respect to the four criteria: ease of learning, ease of use, completeness, and ease of implementation. Finally, a decision is made as to what type of language is best for the current application, and a specification based on this type is described.

Definition of the Algebra and Calculus

Relational Algebra. The representation of the database as a collection of relations encourages the use of set operations for data manipulation. The primary purpose of a relational algebra is to provide a collection of these operations suitable for selecting data from a relational database. Data selection is viewed as the formation (by some operation of the algebra) of a new relation from the existing collection of relations. In addition to the set operations such as the union, intersection, and relative complement, relational operations are also necessary for flexible manipulation of the data. The relational operations include projection onto some domains of a relation, selection of a subset of a relation, join and division of distinct relations (Ref 9: 68, 6: 329).

Relational Calculus. Retrieval languages based on the relational calculus, a version of propositional calculus which describes expressions involving information modeled as relations, are another facility made possible by the use of the relational model. An important aspect of

these languages is that algorithms which generate the relational expression can be described using the set and relational operations. Thus any legal expression represents a relation which can be generated from the information in the other relations of the database.

Example. As an example, the calculus and algebraic representations of the simple query, "Find the names of all suppliers who supply part 43," are given below.

The calculus expression is:

```
RANGE sp x
GET supplies43(s.sname): ∃x(x.s# = s.s# ∧ x.p# =  43 )
```

This expression describes tuples containing supplier names from the s relation into the workspace relation supplies43. The name x is a tuple-valued variable used to relate the two portions of the selection expression. A functionally equivalent set of algrebraic commands for this query is:

```
SELECT ALL FROM sp WHERE p# = 43 GIVING sells43
JOIN sells43, s WHERE s# = s# GIVING temp
PROJECT temp OVER sname GIVING supplies43
```

Algebra vs. Calculus

Ease of Learning. Since the database is being designed for use in an educational environment, the student's ability to quickly grasp the fundamentals of the language is very important in that the less time spent learning and the more time spent using the language will allow the student to gain the most benefit from the experience. Mathematicians consider the relational calculus to be more natural for users, because it allows retrieval of data based on the properties of that data, whereas algebraic manipulation requires the user to specify an algorithm with the algebraic operations necessary to perform the retrieval. Unfortunately,

the quantifiers and bound variables inherent in the propositional calculus make the relational calculus uncomfortable and unnatural for non-mathematicians (Ref 10:23). Since most students interested in the database field have backgrounds which relate more to the algorithmic approach to programming rather than the non-procedural approach of the propositional calculus, the relational algebra is superior to the relational calculus when considering ease of learning.

Ease of Use. It is not always the case that facilities which are easy to learn are easy to use. However, in this case, the relational algebra is easier to use, since the population of users will for the most part be students. The relational algebra works with relations as a whole, rather than tuple-by-tuple as the calculus does. It is easier for the student familiar with traditional programming techniques to take a collection of relations and reduce them in a step-by-step fashion to a resultant set of relations, than to try to formulate a single predicate expression in the relational calculus to describe the result he or she wants.

Completeness. Codd (Ref 9) showed that both the relational algebra and the relational calculus are relationally complete languages; that is, given any finite collection of relations R1, R2, . . ., Rn in simple normal form, the expressions of the algebra or calculus permit definition of any relation definable from R1, R2, . . ., Rn. From the user's viewpoint this means any arbitrary simple or complex question concerning the set of relations in the database can be answered by formulation of a query in either the calculus or algebra. Thus each language has a powerful and basic selective power.

In most practical environments this power needs to be enhanced with the introduction of a capability for invoking any of a finite set of library functions while staying within the algebraic or calculus framework. Codd states that such enhancements readily fit into the calculus framework; but in the algebraic framework, the functions have to be recast in the form of mappings from relations to relations, and this gives rise to circumlocutions. However, there is no reason why library functions cannot be applied just as simply in the algebra as in the calculus. One example is the language SEQUEL (Ref 11) which uses an algebraic framework and includes such functions as COUNT, SUM, MAX, MIN, etc. Thus both the algebra and the calculus satisfy the requirement of completeness.

Ease of Implementation. It was noted that the algebra is somewhat more procedural in comparison with the calculus. For the Data Base Management System (DBMS) this procedurality can be an advantage in that implementation, on one level at least, can be reasonably straightforward: the DMBS can simply perform all the joins, projections, and other operations as specified in the expression that the user has written. On the other hand, such an implementation would not be very efficient, and would very likely result in the user having to expend time and effort in choosing the most efficient expression of the query -- clearly an undesirable state of affairs. However, as will be pointed out in the next section, work has been done on optimizing the implementation of algebraic expressions (Ref 12); and thus this objection may cease to be valid if sufficiently efficient schemes are developed and implemented (Ref 2:120).

There are basically two reasons why a relational algebra is a better implementation of an appropriate optimization interface : (1) A relational algebra treats and manipulates whole relations as single

objects whereas the relational calculus type of interface deals with the relations on a tuple-by-tuple basis. A relational algebra may therefore be considered to be at a higher level of abstraction than these other interface systems, and thus offer more scope for high level optimization. (2) If a relational algebra is conducive to smart optimization, it may provide a practical implementation level for other query languages. Indeed, Codd (Ref 9) has developed an algorithm for supporting a relational calculus over a relational algebra (Ref 12:569).

Description of the Relational Algebra Based Design

For the type of system being developed, the above arguments tend to favor the relational algebra approach. Thus a data manipulation and definition language was designed based on the relational algebra. The specification for the language appears in Appendix B.

The data manipulation language includes all the traditional relational operations as discussed in the definition section. In addition, statements are provided for insertion, deletion and modification of the tuples in a relation, the ability to attach stored relations, and delete, save, copy, sort and rename those relations or relations created by means of the relational operations. Statements are also provided for preparing a relation for report generation and for inputting bulk data into a relation. Note that particular forms of the SELECT statement do provide for functions in the query (Refer to Completeness subsection).

The data definition language includes commands to define domains and define relations based on those domains. As part of the definition of a relation, security controls and integrity constraints are specified. These are important for protecting an individual's data from

being altered or deleted without his or her knowledge. Defined in this way security controls apply to an entire relation. However, in the future, it may be necessary (due to Privacy Act or "need to know" constraints) to have security on a tuple-by-tuple basis. In this case one or more attributes of a relation would be declared as security attributes which are never displayed but must be specified when performing the otherwise forbidden operation on the particular tuple.

All overall security controls may be optionally specified with the exception of the ID control. This identifies the owner or creator of the relation and allows the database administrator to recognize these owners. In addition the owner is allowed to perform any operation on his or her relations and to change the passwords associated with the security controls on his or her relations. In essence the ID security control allows each user to be his own database administrator. This allows the student to gain experience in database administration as well as manipulation of the data. The administrator of the entire collection of databases would typically be the instructor or a high level manager. He or she would be allowed to access, change, or delete any relation in anyone's database in addition to performing general maintenance of the system, such as changing the storage algorithms to improve efficiency.

Summary

Most database languages for relational systems fall into a relational calculus or a relational algebra framework. It was shown that in light of the criteria -- ease of learning, ease of use, completeness, and ease of implementation -- and in light of the goal of a pedagogical system, the relational algebra was superior. This was then used as a

basis for the design of the database manipulation and definition language appearing in Appendix B.

SOLUTIONS TO THE PROBLEMS OF RELATIONAL DATABASES

Determining methods to eliminate or reduce the problems of relational systems has occupied many researchers' time for nearly a decade. Work has progressed slowly in this area because the relational model was slow to gain acceptance as a viable alternative to the more firmly entrenched hierarchical and network models. This section presents the methods used in this thesis to alleviate the problems described in the first section of this chapter, especially the problem of inefficiency.

Integrity

Integrity deals with the prevention of semantic errors made by users due to their carelessness or lack of knowledge. The integrity problems mentioned in the Flexibility subsection have not been entirely solved due to the limited scope of this thesis. However, in an excellent paper Eswaran and Chamberlin (Ref 13) have laid out the functional specifications of a subsystem for database integrity. All or parts of this subsystem could be easily added to an already existing modular database system.

This subsystem permits users to make assertions which define the "correctness" of the database, and to specify actions to be taken when the assertions are not satisfied. There are both static and dynamic integrity assertions. Static assertions are constraints which must hold true for the life of the data object. For example, part numbers may be required to be non-negative integers, and any insertion or modification which would create a negative part number would be rejected.

Dynamic integrity assertions may be added and dropped from time to time, and may describe not only the nature of a data object but also its relationship to other objects. This is the type of assertion which would be needed to solve the problem with the example of the network model, where a relation of edges and a relation of nodes have implicit relationships between the tuples. This thesis has only developed the structure for using static integrity assertions in the definition of domains and relations (See Appendix B).

Redundancy of Data and Efficiency at the Storage Representation Level

Data compression techniques are one way to remove redundancy of data. Another, more powerful method, involves combining the indexes of more than one relation into a common structure. In the User Expectations subsection, the problems of efficiency of update vs. efficiency of retrieval were pointed out. Some implementations have tried to solve this problem by creating two access structures -- a link structure among relations to provide efficient retrieval, and an image structure for individual relations to provide efficient update. This method has several drawbacks. One is that it really doesn't help the data redundancy problem since many of the keys have to exist in both access structures. In addition, the system must support two different sets of access procedures.

In a recent paper (Ref 15), Theo Haerder has developed a method for combining the two structures into a generalized access path structure. In order to explain Haerder's structure, some background material on images and links will be presented, and then a method for combining these two structures and its generalization will be described.

Definitions. An access path giving value ordering and associative access by one or more attributes to one relation is called an "image". For example, using the parts-supplier model introduced in Chapter I, an image on the CITY attribute of the PART relation would provide access to tuples based on the specification of a value for CITY. A value of "LONDON" would cause the access method to return the values "P1" and "P4", or appropriate pointers to those two tuples.

Definition -- Let R be a relation with attributes $A_1, \ldots, A_n$. An image I_i of the attribute A_i of R, $i \in \{1, \ldots, n\}$, is a mapping from values in A_i to those tuples in R which have that value for the ith attribute. Additionally, these sets of tuples qualified by values of A_i are ordered according to the sorted sequence of values of A_i. The generalization of the term "image" to compound attributes is straightforward.

Access paths relating tuples of one relation to tuples of another relation are called binary links. For example, a binary link between the tuples of the PART and SP relations would provide an access path from the tuples of PART to the tuples of SP where the P# attributes were the same. Thus, a value of P3 from the PART relation would cause the access method to return the value (S1,P3), or an appropriate pointer to this tuple. Haerder uses special binary links according to the following definition.

Definition -- Let R be a relation with attributes $A_1, \ldots, A_n$, S be a relation with attributes $B_1, \ldots, B_m$; $F(A_i) = F(B_n)$ for the domains $F(A_i)$, $F(B_k)$, $i \in \{i, \ldots, n\}$, $k \in \{1, \ldots, m\}$; A_i be a candidate key of R. The link between R and S with regard to A_i, B_k is defined as the set $L(R(A_i), S(B_k)) := \{(r,s)/r \in R, s \in S, pr_{A_i}(r) = pr_{B_k}(s)\}$, where $pr_{A_i}(r)$ and $pr_{B_k}(s)$ are the projections to the components of r and s which correspond to attributes A_i and B_k, respectively. The term "link" may be generalized for compound attributes similarly.

The reference from the access path structure to the actual tuple is usually done by means of TID's (Tuple Identifiers) or physical pointers. An appropriate implementation technique for TID's is a concatenation of a page number along with a byte offset from the bottom of that page. This combines the speed of a byte address pointer with the flexibility of indirection. The page number allocated in a logical address space allows an indirect reference to the actual physical storage block. The offset denotes a special slot which contains the byte location of the referenced tuple in the page. Hence, the TID concept offers two different kinds of indirection -- at the page level and within the page.

Implementation Technique for Images. An image is conveniently implemented and maintained through the use of a multipage index structure which contains pointers to the tuples themselves. The pages of a given index can be organized into a balanced hierarchic structure using the concept of B*-trees (pronounced B-star trees) (Ref 16,17). For non-leaf nodes, an entry consists of a key value and a pointer pair. The key itself can consist of values of single or compound attributes and can be represented in encoded form (Ref 18) allowing a particular sort

order on each attribute value in case of compound attributes. The pointer addresses another nonleaf page or a leaf page of the same structure.

For the leaf nodes an entry is a combination of key values, along with a variable length ascending list of TID's for tuples having exactly those key values. In order to identify the length of the TID list an additional length information field is kept with each stored key. In addition, the leaf pages are chained in a doubly linked list, so that sequential access can be supported from leaf to leaf.

If the total storage space for the TID lists of a particular key exceeds one leaf page, overflow pages can be introduced optionally which can hold the overflowing part of the lists. These overflow pages are chained with the leaf pages only, and they are not pointed to by the nonleaf pages, in order to reduce the increase of the height of the B*-tree.

If a mechanism is provided for enforcing the uniqueness of keys, this structure can also be used to implement an access path for primary keys. The "image" of the relation is represented by the particular value ordering when accessing the leaves of the B*-tree from left to right (in post order). When a relation is created, one image of the relation may be designated as the "clustering image," with the result that tuples near each other according to a chosen order relation will be stored physically near by.

Figure 3 shows schematically an image on the attribute CITY of the PART relation. Assume that only the first four of many tuples in the PART relation are given in Figure 1(c). Key values; e.g., "P2", are used as tuple identifiers (TID's) and their use as such is indicated by enclosing them in parenthesis. The image as shown on the attribute CITY is denoted by I(PART(CITY)).

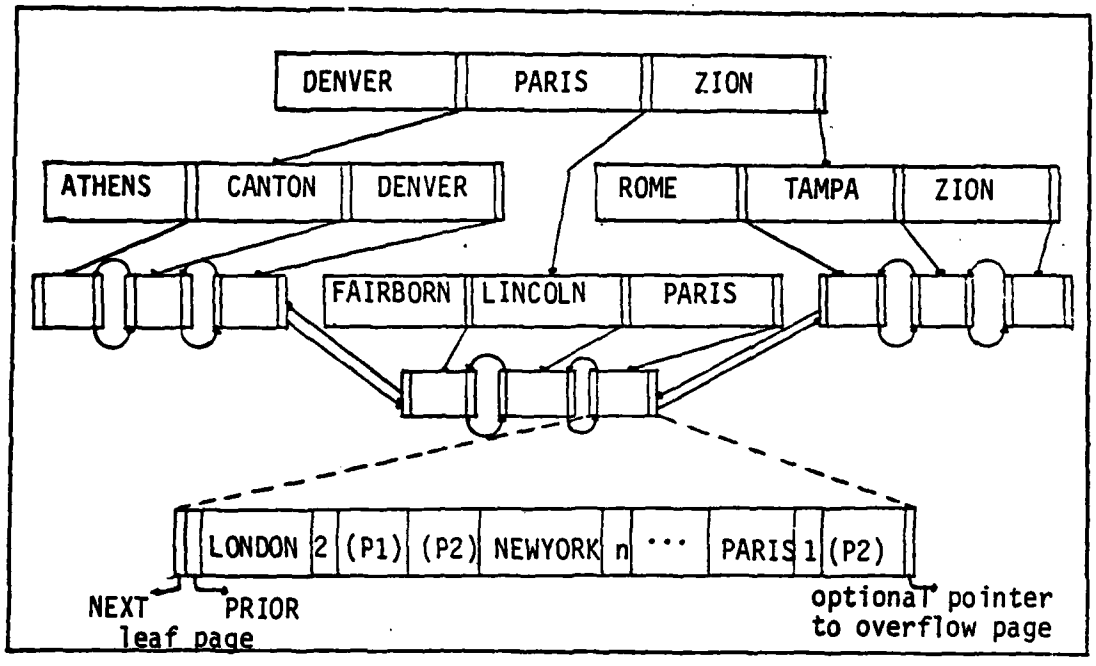


Figure 3. Image implementation for I(PART(CITY)).

Implementation Technique for Links. A binary link connects tuples in one or two relations on matching attribute values. Usually, it is implemented by using chaining techniques with TID's or physical pointers (storage addresses). The TID chaining gives one level of indirection compared to physical chaining of addresses.

For example, links are maintained in the Relational Storage System by storing the TID's of the NEXT, PRIOR, and OWNER tuples in the prefix of the child tuples and by storing at least the TID of the first child tuple in the parent tuple according to Figure 4. In this example one tuple of the OWNER relation PART(P#, . . .) is linked to 3 tuples of the MEMBER relation SP(S#,P#,QTY). The binary link is denoted by L(PART(P#),SP(P#)).

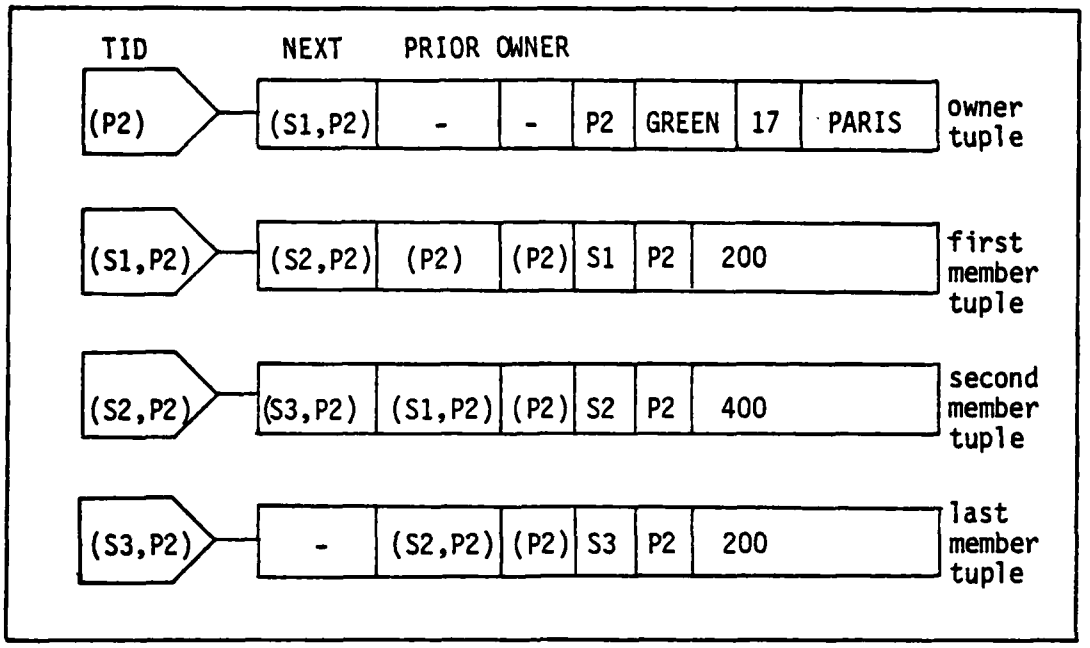


Figure 4. Link implementation of L(PART(P#),SP(P#)); link occurrence for domain value P2.

Implementation Technique for a Combined Access Path Structure. A binary link provides a direct path from single tuples (parents) in one relation to sequences of tuples (children) in another relation. Usually it is argued that the main advantage of a link is the direct access to a tuple of either relation coupled by a binary link, while use of an image may involve a complete traversal of a B*-tree structure consisting of several page accesses in order to find the child or parent tuple. The relative gain of a link over an image is even enhanced when the child tuples have been clustered on the same page as the parent tuple. In this case no additional page has to be touched using the link, while a couple of pages may be accessed in a large index.

It should be pointed out the relationships between tuples of one or different relations are expressed explicitly by attribute values in the relational model. This key property allows combined images on

the same domain serving also as link structures. Therefore, the advantage of image and link access can be combined using a different kind of organization of the leaf nodes of the B*-tree. The nonleaf nodes look exactly as in the single image implementation. In the leaf nodes, separate TID lists for both relations together with the related length information fields are stored for each key. The lists for the parent relation contain only one TID entry, while each variable length list for the child relation contains the sequence of TID's for the children related to a particular parent tuple. The order in these lists can be exactly the same as in the binary link. In Figure 5 the discussed examples for the SP and PART relations are treated in a unified way. The various attribute values for P# in SP and PART are the keys in the images and the matching P#'s also establish the link occurrences between the two relations.

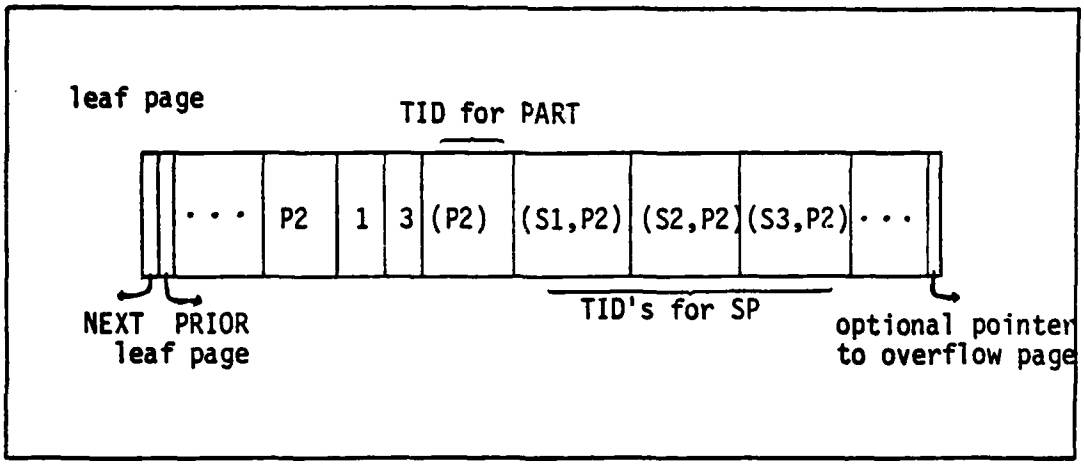


Figure 5. Combined implementation of link L(PART(P#),SP(P#)), and the images I(PART(P#)) and I(SP(P#)).

With this access path structure the striking disadvantage of separate images can be avoided, that is, the traversal of an additional B*-tree structure, when the child tuples are to be accessed after the parent tuple is located. In either case it must be assumed that the owner tuple is found via an image access I(PART(P#)). If the leaf page containing the required key (candidate key) for the tuple of the OWNER relation is fixed in core, then the subsequent navigational accesses to the tuples of the MEMBER relation are at least as fast as the accesses via the binary link. In case of clustering, even more tuples can be stored in a particular page, because the storage space of three TID's per tuple and link is saved. On the other hand, the access to the linked tuples in determined sequence enforced by the embedded TID chain is not necessary. Furthermore, having the combined access path structure, there is no need to fetch the tuples of a binary link sequentially; e.g., if it happens that the tuples are stored on different devices, seeks and rotational delays may be overlapped.

Generalization of the Combined Access Path Structure. The combined access path structure replaces different access path types like image and binary link by joining the various characteristics of these access paths in one unified structure. A considerable advantage is gained, therefore, with regard to implementation complexity. Instead of supporting specialized modules for each of the access path types, only one unified set of modules working on this combined structure is necessary. The proposed approach reduces the extent of implementing various operations on access paths.

The proposed concept of the combined access path structure can be extended in the following way leading to the "generalized access path

structure": All variable length TID lists belonging to the various attributes in different relations which are all defined on the same domain are stored with their related domain value (key value). This concept is not restricted to a single domain with single attributes defined on it. It can be applied to given sequences of attributes (compound attributes) corresponding to one particular domain sequence.

The format of the nonleaf pages is the same as for the image and combined access path. All kinds of optimizations; e.g., key compression, which are available for single access path implementation can be applied to them. (See especially "Prefix B-Trees" by Bayer and Unterauer (Ref 18).) The leaf pages contain for each key up to m variable length TID lists together with m length information fields. If an actual domain value is not defined for attribute A_i, then the corresponding TID list does not exist and the corresponding length information field indicates this fact by having a zero entry. At least one TID list must exist for a specified domain value; otherwise the domain value is currently not used in any tuple of the related relations and doesn't appear as a node in the access path.

The implementation of the generalized access path structure is shown in Figure 6. The particular example is chosen for four relations related by domain PART NUMBER. Let us assume that the relation R1 is PART with P# being the primary key. R2 may be considered as the SP relation with the inverted attribute P#. R3 and R4 are introduced as the MANAGER and EQUIPMENT relations:

MGR(M#,P#,JCODE,...)
EQUIP(INO,P#,TYPE,...).

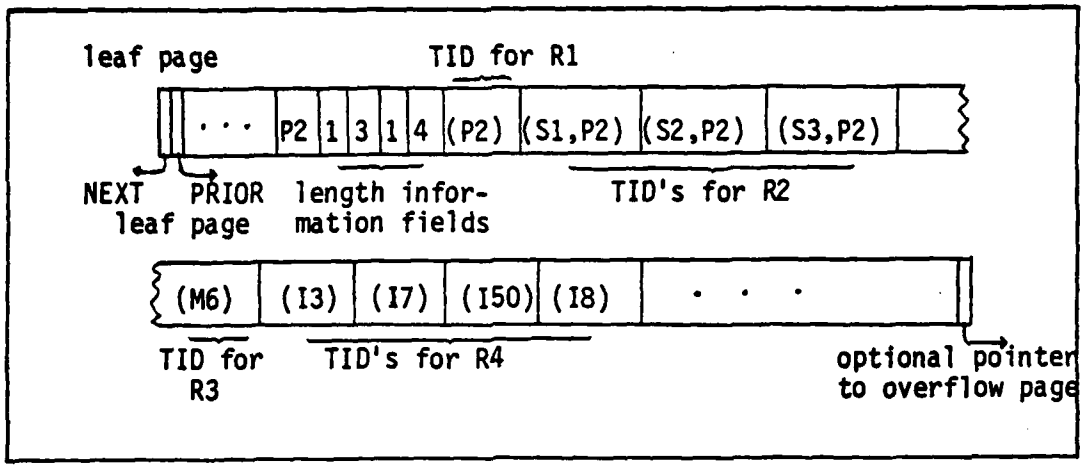


Figure 6. Implementation of the generalized access path; example for four relations on domain P#.

The attributes P# of the relations MGR and EQUIP are also inverted. P# in relation MGR is specified as a candidate key, additionally. The graphical representation of this example describing all existing binary links in it is shown in Figure 7.

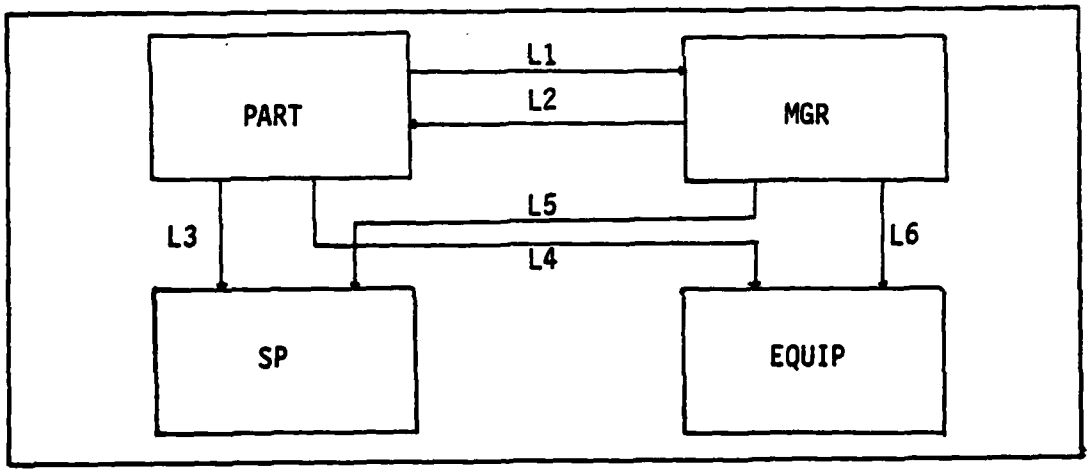


Figure 7. All existing binary links based on P#.

Here the same attribute name P# is chosen for convenience. In principle each relation can have a different attribute name defined on the same domain; e.g., PART NUMBER. In the case of domains with numeric values each attribute can carry a different unit of the same or different unit types. By accessing the index the appropriate conversion rule must be applied to map the particular attribute value to the corresponding domain value.

Each node in the leaf page; e.g., the particular node with domain value "P2" in Figure 6, contains four variable length lists with four length information fields describing the tuples of the four relations with P# = "P2". If a particular attribute is specified as a candidate key, the corresponding list length of the TID list is restricted to 1, shown in the example for domain value "P2" for R1 and "M6" for R3. All other attributes are not restricted at all.

Summary. In summary a generalized access path structure combines the advantages of link and image structures in retrieval and update operations, and is competitive from a performance point of view. In addition, the various kinds of concievable pointers such as FIRST, NEXT, PRIOR, OWNER, etc., can be represented by their relative position in the variable length TID list. As a result a substantial saving of storage space is gained with this structure. Finally, this unified approach to access path implementation should reduce the complexity of the system implementation.

Efficiency at the Conceptual Level

Relational database systems provide the user with a tabular view of the data, a view that is independent of any machine or implementation.

The user need know nothing of the implementation in formulating his query. Unfortunately, because a user is deliberately made unaware of the actual data storage mechanism, he may write queries which, though consistent with his relational view, have a very low efficiency factor. It is essential that the burden of efficiency, since effectively removed from the user, be assumed by the interface to the database.

The previous section presented methods for removing redundancy and improving efficiency at the storage representation level. In addition, very significant optimization can be done at higher levels of interpretation where the global structure of a user query is known. Both Smith and Chang (Ref 12) and Hall (Ref 14) have designed some algorithms and heuristics to use in optimizing single expressions for a relational algebra interface.

The basic organization of the optimizer is shown in Figure 8. Syntax analysis entails checking the input query for proper form, ensuring that security constraints are not being violated, and creating an operator tree. This tree is then passed to the tree transformer which has access to a set of correctness-preserving algebraic transformations (Tables I-IV), and also to a set of rules which determine when the application of these transformations will increase efficiency.

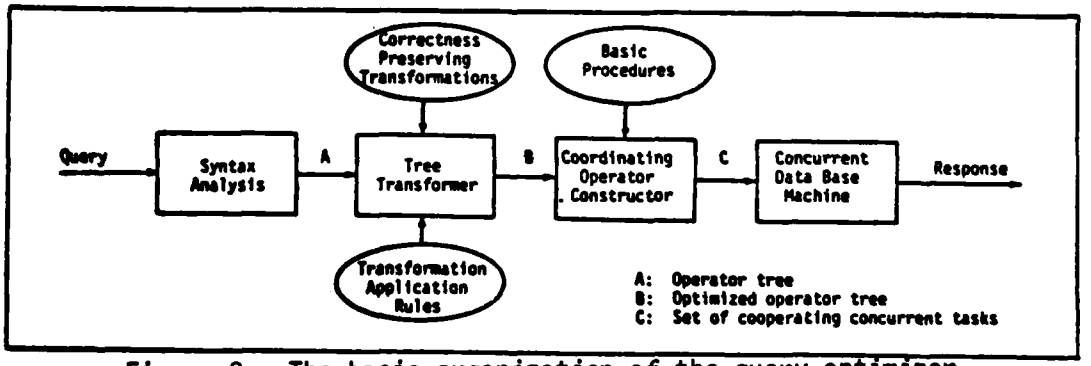


Figure 8. The basic organization of the query optimizer.

The transformer optimizes the tree and passes it along to a mechanism which constructs an implementation of each operator as a task. This operator has access to a set of basic implementation procedures (Appendix C). The constructor creates tasks from these procedures in such a way that the performance of the whole tree of cooperating tasks is optimized. This is achieved by distributing and analyzing the effects on sort order of possible implementation decisions, and then creating tasks so as to coordinate sort order throughout the task tree. Smith and Chang provide a two pass procedure to implement this process: the first pass up and second pass down the tree.

On the upward pass each branch is labeled with the set of sort orders which can be efficiently generated from lower operations. These are called preferred sort orders (pso). Then a pass is made down the tree. As one goes down, the sort orders which can be most efficiently supplied from below to a given operator node are already known, and thus the sort order this node must pass up can also be determined. So a pso is chosen from below while simultaneously constructing the appropriate implementation of the operator. Tuples will be re-sorted at a node only if no pso from below can be effectively utilized. The UP-rules for the upward pass are presented in Table V, and the DOWN-rules for the downward pass are presented in Table VI.

The final step is to execute the procedures. Although impossible with the configuration assumed for this thesis, the most advantageous implementation would run the procedures on a multiprocessing machine in order to exploit the concurrency among tasks, generated by the above procedures. These procedures are described in more detail in Chapter IV.

SUMMARY

Relational systems have significant advantages over other systems including simplicity, generality and flexibility. However, a large and overpowering disadvantage in any simple implementation of a relational system is its lack of efficiency. By choosing a relational algebra framework for the DML and DDL, the pedagogical needs of the system are met, as well as opening up the way for implementation of the high level optimization techniques discussed in the last section. The efficiency problem can also be abated at a low level with the use of the generalized access path structure also described in the last section. The methods used to implement these procedures are the topics of the next chapter.

Table I. Idempotency Laws of Relational Algebra

Expression	Reduces To
1. $A \cup A$	A
2. $A \cap A$	A
3. $A - A$	$\emptyset$
If $A \subset B$ then	
4. $A \cup B$	B
5. $A \cap B$	A
6. $A - B$	$\emptyset$

Table II. Simplification Laws of Boolean Algebra

Expression	Reduces To
1. $a \wedge a$	a
2. $a \vee a$	a
3. $a \vee (a \vee b)$	$a \vee b$
4. $a \wedge (a \vee b)$	a

Table III. Distribution Laws for Moving SELECTs Down Trees

Operation	Expression	Transformed To	Alternative
selection :	(A : F) : G	A : (F and G)	
union ∪	(A ∪ B) : F	(A : F) ∪ (B : F)	
intersection ∩	(A ∩ B) : F	(A : F) ∩ (B : F)	(A : F) ∩ B, A ∩ (B : F)
difference -	(A - B) : F	(A : F) - (B : F)	(A : F) - B
divide ÷	(A ÷ B) : F	(A : F) ÷ B	
projection '/.	(A '/. T) : F	(A : F) '/. T	
join, product *	(A * B) : F	((A : F1) * (B : F2)) : F3	

where F = F1 and F2 and F3, see Chapter IV

Table IV. Distribution Laws for Moving PROJECTs Down Trees

Operation	Expression	Transformed To	Application Rule
selection :	(A : F) ⅋. T	(A ⅋. T) : F	- only if SELECT not at a leaf and attributes of F are in T.
union ∪	(A ∪ B) ⅋. T	(A ⅋. T) ∪ (B ⅋. T)	- always
intersection ∩	(A ∩ B) ⅋. T	(A ⅋. T) ∩ (B ⅋. T)	- only if T contains primary keys of A if A a stored relation and of B if B a stored relation.
difference -	(A - B) ⅋. T	(A ⅋. T) - (B ⅋. T)	- same as intersection.
join, product *	(A * B) ⅋. T	((A ⅋. T1) * (B ⅋. T2)) ⅋. T3	- see Chapter IV.
projection ⅋.	(A ⅋. T) ⅋. U	A ⅋. U	- always

Table V. Rules for coordinating sort order and implementing operators -- UP-rules (Ref 12).

Notation:

d_R is either the domain which R is sorted on (if R is sorted) or n (if R is not sorted).

γ_R Is the preferred sort order (pso) set for Relation R. N.B. γ_R may also include n.

I_R, where R is a stored relation, is the set of domains in R for which directories (Indices) exist.

ϕ is the empty set.

KEY(R) is the set of primary key domains in R.

UNARY(R) is true if R consists of a single domain.

	LEAF NODE		INTERNAL NODE	
OPERATION	INPUT LABEL	OUTPUT LABEL	INPUT LABEL	OUTPUT LABEL
R[β]	d_R	*if* ($d_R \notin \beta$ *or* KEY(R)$\subseteq\beta$) *then* (d_R) *else* β	γ_R	*if* $\beta \cap \gamma_R \neq \phi$ *then* $\beta \cap \gamma_R$ *else* β
R[E]	d_R	(d_R)	γ_R	γ_R
R[C=D]S	d_R d_S	*if* UNARY(R) *and* (D$\in I_S$ *or* D$\neq d_S$) *then* (d_S) *else if* UNARY(S) *and* (C$\in I_R$ *or* C$\neq d_R$) *then* (d_R) *else* (C,D)	γ_R γ_S	*if* UNARY(R) *and* D$\in\gamma_S$ *then* γ_S *else if* UNARY(S) *and* C$\in\gamma_R$ *then* γ_R *else* (C,D)
R[CθD]S ($\theta \neq$ "=")	d_R d_S	*if* C=d_R *and* D$\neq d_S$ *then* (d_S) *else if* C$\neq d_R$ *and* D=d_S *then* (d_R) *else* ($d_R d_S$)	γ_R γ_S	*if* C$\in\gamma_R$ *and* D$\in\gamma_S$ *then* γ_S *else if* C$\in\gamma_R$ *and* D$\in\gamma_S$ *then* γ_R *else* $\gamma_R \cup \gamma_S$
R$\cup$S	d_R d_S	*if* d_R=$d_S \neq n$ *then* (d_R) *else* (n)	γ_R γ_S	*if* $(\gamma_R \cap \gamma_S)-(n) \neq \phi$ *then* $(\gamma_R \cap \gamma_S)-(n)$ *else* (n)
R$\cap$S	d_R d_S	($d_R d_S$)	γ_R γ_S	*if* $(\gamma_R \cap \gamma_S)-(n) \neq \phi$ *then* $(\gamma_R \cap \gamma_S)-(n)$ *else* $\gamma_R \cup \gamma_S$
R-S	d_R d_S	*if* $d_R \neq d_S$ *and* $I_R=\phi$ *then* (n) *else* (d_R)	γ_R γ_R	*if* $(\gamma_R \cap \gamma_S)-(n) \neq \phi$ *then* $(\gamma_R \cap \gamma_S)-(n)$ *else* γ_R
R$\times$S	d_R d_S	($d_R d_S$)	γ_R γ_S	$\gamma_R \cup \gamma_S$
R[$\alpha \ddagger \beta$]S	d_R d_S	*if* $d_R \in \alpha$ *then* d_R *else* α	γ_R γ_S	*if* $\gamma_R \cap \alpha \neq \phi$ *then* $(\gamma_R \cap \alpha)$ *else* α

	LEAF/INTERNAL NODE		INTERNAL/LEAF NODE	
OPERATION	INPUT LABEL	OUTPUT LABEL	INPUT LABEL	OUTPUT LABEL
R[C=D]S	d_R γ_S	*if* UNARY(R) *and* D$\in\gamma_S$ *then* γ_S *else if* UNARY(S) *and* (C$\in I_R$ *or* C$\neq d_R$) *then* (d_R) *else* (C,D)	γ_R d_S	SYMMETRICAL
R[CθD]S ($\theta \neq$ "=")	d_R γ_S	*if* C=d_R *and* D$\in\gamma_S$ *then* γ_S *else if* C$\neq d_R$ *and* D$\in\gamma_S$ *then* (d_R) *else* (d_R)Uγ_S	γ_R d_S	SYMMETRICAL
R$\cup$S	d_R γ_S	*if* $d_R \neq n$ *and* $d_R \in \gamma_S$ *then* (d_R) *else* (n)	γ_R d_S	SYMMETRICAL
R$\cap$S	d_R γ_S	*if* $d_R \neq n$ *and* $d_R \in \gamma_S$ *then* (d_R) *else* (d_R)Uγ_S	γ_R d_S	SYMMETRICAL
R-S	d_R γ_S	(d_R)	γ_R d_S	*if* $d_S \neq n$ *and* $d_S \in \gamma_R$ *then* (d_S) *else* (n)
R$\times$S	d_R γ_S	(d_R)Uγ_S	γ_R d_S	SYMMETRICAL
R[$\alpha \ddagger \beta$]S	d_R γ_S	*if* $d_R \in \alpha$ *then* d_R *else* α	γ_R d_S	*if* $\gamma_R \cap \alpha \neq \phi$ *then* $(\gamma_R \cap \alpha)$ *else* α

Table VI. Rules for coordinating sort order and implementing operators -- DOWN rules (Ref 12).

Notation:

L: left operand branch (binary) / operand branch (unary)
R: right operand branch (binary)

$a(\gamma)$ is an arbitrary element of γ.
$a'(\gamma)$ is an arbitrary element of $(\gamma - \{a\})$.
r(D) is the relation to which domain D belongs
C and {C} are used interchangeably.
DIREC[R,D] creates a directory for relation R over domain D.

PIPE S = $[U \leftarrow R; R \leftarrow \gamma \cap \gamma_R]$
PIPE R = $[U \leftarrow S; L \leftarrow \gamma \cap \gamma_R]$

If no input label change is specified then the label remains the same.

	LEAF NODE		INTERNAL NODE	
OPERATION	OUTPUT LABEL	OPERATOR BINDING	OUTPUT LABEL	LABEL & OPERATOR BINDING
R[$\mathcal{A}$]	γ	*if* $\mathcal{A}=\gamma_R$ *then* PROJECT1[R,$\mathcal{A}$] *else if* $\mathcal{A}\neq\mathcal{A}$ *then* PROJECT2[R,$\mathcal{A}$] *else* PROJECT3[R,$\mathcal{A}$,γ]	γ	*if* $\mathcal{A}\cap\gamma_R\neq\emptyset$ *then* [$L\leftarrow(\mathcal{A}\cap\gamma)_L$ PROJECT2[R,$\mathcal{A}$]] *else* PROJECT3[R,$\mathcal{A}$,γ]]
R[E]	d_E	SELECT[R,E]	γ	$L\leftarrow\gamma_L$ SELECT[R,E]
R[C=D]	γ	*if* UNARY(R) *and* (D$\in I_R$ *or* D=d_R) *then* JOIN1[R,C,=,D,S,R] *else if* UNARY(S) *and* (C$\in I_S$ *or* C=d_S) *then* JOIN1[R,C,=,D,S,S] *else* JOIN2[R,C,=,D,S]	γ	*if* UNARY(R) *and* D$\in\gamma_R$ *then* {JOIN1[R,C,=,D,S,R]; $R\leftarrow\gamma$} *else if* UNARY(S) *and* C$\in\gamma_R$ *then* {JOIN1[R,C,=,D,S,S]; $L\leftarrow\gamma$} *else* {*if* C$\in\gamma_R$ *then* $L\leftarrow S$; *if* D$\in\gamma_R$ *then* $R\leftarrow D$; JOIN2[R,C,=,D,S]}
R[C<D] (D ≠ "=")	γ	*if* C=d_R *and* D=d_S *then* U←[R,S]-r(a'(γ)) *else if* C=d_R *then* U←R *else if* D=d_S *then* U←S *else if* γ=[n] *then* U←R *else* U←[R,S]-r(a'(γ)); JOIN0[R,C,R,D,S,U]	γ	*if* C$\in\gamma_R$ *and* D$\in\gamma_S$ *then* {*if* a(γ)$\in\gamma_R$ *then* {*PIPE* S_1 R←D} *else* {*PIPE* S_1 L←C}} *else if* C$\in\gamma_R$ *then* {*PIPE* S_1 L←C} *else if* D$\in\gamma_S$ *then* {*PIPE* S_1 R←D} *else if* a(γ)$\in\gamma_S$ *then* *PIPE* R *else* *PIPE* S_1; JOIN0[R,C,R,D,S,U]
R∩S	γ	*if* $d_R=d_S$ *then* UNION1[R,S] *else if* $I_R\cap I_S\neq\emptyset$ *then* UNION2[R,S] *else if* $d_R\in I_S$ *or* $d_S\in I_R$ *then* UNION3[R,S] *else* UNION4[R,S]	γ	*if* ($\gamma_R\cap\gamma_S$)-[n]≠∅ *then* {E←a'(γ)$_L$ L←E_1 R←E_2 UNION1[R,S]} *else if* $\gamma_R\neq$[n] *then* {E←a'(γ_R)$_L$ L←E_1 DIREC[S,E] UNION3[R,S]} *else if* $\gamma_S\neq$[n] *then* {E←a'(γ_S)$_R$ R←E_2 DIREC[R,E] UNION3[R,S]} *else* UNION4[R,S]
R∩S	γ	*if* $d_R=d_S$ *then* INTER1[R,S] *else if* $I_R\cap I_S\neq\emptyset$ *then* INTER2[R,S,r(a'(γ))] *else if* $d_R\in I_S$ *or* $d_S\in I_R$ *then* INTER3[R,S,r(a'(γ))] *else* INTER4[R,S,r(a'(γ))]	γ	*if* ($\gamma_R\cap\gamma_S$)-[n]≠∅ *then* {E←a'(γ_R)$_L$ L←E_1 R←E_2 INTER1[R,S]} *else if* $\gamma_R\neq$[n] *then* {E←a'(γ_R)$_L$ *if* E∈I_S *then* {L←E_1 DIREC[S,E] INTER3[R,S,R]} *else* {R←E_2 DIREC[R,E] INTER3[R,S,S]}} *else* INTER4[R,S,γ_R]
R-S	γ	*if* $d_R=d_S$ *then* DIFF1[R,S] *else if* $I_R\cap I_S\neq\emptyset$ *then* DIFF2[R,S] *else if* $d_R\in I_S$ *or* $d_S\in I_R$ *then* DIFF3[R,S] *else* DIFF4[R,S]	γ	*if* ($\gamma_R\cap\gamma_S$)-[n]≠∅ *then* {E←a'(γ_R)$_L$ L←E_1 R←E_1 DIFF1[R,S]} *else if* $\gamma_R\neq$[n] *then* {E←a'(γ_R)$_R$ R←E_2 DIREC[R,E] DIFF3[R,S]} *else* DIFF4[R,S]
R×S	γ	*if* $d_R\in\gamma$ *then* CARPROD[R,S,R] *else* CARPROD[R,S,S]	γ	*if* $\gamma\cap\gamma_R\neq\emptyset$ *then* {L←$\gamma\cap\gamma_R$ CARPROD[R,S,R]} *else* {R←$\gamma\cap\gamma_S$ CARPROD[R,S,S]}
R[$\alpha\leq\beta$]S	γ	$\gamma\leftarrow S[\mathcal{A}]$ *if* $d_R\in\mathcal{A}$ *then* DIVIDE[R,α,$\mathcal{A}$,T,90] *else* DIVIDE[R,α,$\mathcal{A}$,T,a($\mathcal{A}$)]	γ	$\gamma\leftarrow S[\mathcal{A}]$ *if* $\gamma_R\cap\mathcal{A}\neq\emptyset$ *then* {L←$\gamma_R\cap\mathcal{A}$; DIVIDE[R,α,$\mathcal{A}$,T,90]} *else* DIVIDE[R,α,$\mathcal{A}$,T,a($\mathcal{A}$)]

DOWN-RULES

	LEAF/INTERNAL NODE		INTERNAL/LEAF NODE	
OPERATION	OUTPUT LABEL	LABEL & OPERATOR BINDING	OUTPUT LABEL	LABEL & OPERATOR BINDING
R[$\mathcal{A}$]	γ	*if* UNARY(R) *and* D$\in\gamma_R$ *then* {JOIN1[R,C,=,D,S,R]; $R\leftarrow\gamma$} *else if* UNARY(S) *and* (C$\in I_S$ *or* C=d_S) *then* JOIN1[R,C,=,D,S,S] *else* {*if* D$\in\gamma_R$ *then* $R\leftarrow D$; JOIN2[R,C,=,D,S]}	γ	SYMMETRICAL
R[C=D]	γ	*if* C=d_R *and* D$\in\gamma_R$ *then* {*if* a(γ)$\in\gamma_R$ *then* *PIPE* S *else* {*PIPE* R_1 R←D}} *else if* C=d_R *then* *PIPE* S *else if* D$\in\gamma_R$ *then* {*PIPE* R_1 R←D} *else if* a(γ)$\in\gamma_R$ *then* *PIPE* S *else* *PIPE* R_1; JOIN0[R,C,R,D,S,U]	γ	SYMMETRICAL
R∩S	γ	*if* $d_R=d_S$ *then* {R←d_R UNION1[R,S]} *else if* $I_R\cap\gamma_S\neq\emptyset$ *then* {R←$I_R\cap\gamma_S$ UNION2[R,S]} *else if* $\gamma_R\neq$[n] *then* {E←a'(γ_R)$_R$ R←E_2 DIREC[R,E] UNION3[R,S]} *else* UNION4[R,S]	γ	SYMMETRICAL
R∩S	γ	*if* $d_R=d_S$ *then* {R←d_R INTER1[R,S]} *else if* $I_R\cap\gamma_S\neq\emptyset$ *then* {E←a'(γ)$_R$ *if* E∈d_R *then* {R←$I_R\cap\gamma_S$ INTER3[R,S,R]} *else* {R←$I_R\cap\gamma_S$; INTER3[R,S,S]}} *else if* $\gamma_R\neq$[n] *then* {E←a'(γ_R)$_R$ *if* E∈d_S *then* {DIREC[R,E] R←E_2 INTER3[R,S,S]} *else* {F←a'(γ_S)$_R$ DIREC[R,F] R←F_2 INTER3[R,S,R]}} *else* INTER4[R,S,R]	γ	SYMMETRICAL
R-S	d_R	*if* $d_R=d_S$ *then* DIFF1[R,S] *else if* $I_R\cap\gamma_S\neq\emptyset$ *then* {R←$I_R\cap\gamma_S$ DIFF3[R,S]} *else if* $\gamma_R\neq$[n] *then* {E←a'(γ_R)$_R$ R←E_2 DIREC[R,E] DIFF3[R,S]} *else* DIFF4[R,S]	γ	*if* $d_R\in\alpha$ *and* $d_R\in\gamma_S$ *then* {L←d_R DIFF1[R,S]} *else if* $I_R\cap\gamma_S\neq\emptyset$ *then* {L←$I_R\cap\gamma_S$ DIFF3[R,S]} *else if* $\gamma_R\neq$[n] *then* {E←a'(γ_R)$_R$ L←E_1 DIREC[S,E] DIFF3[R,S]} *else* DIFF4[R,S]
R×S	γ	*if* $d_R\in\gamma$ *then* CARPROD[R,S,R] *else* {R←$\gamma\cap\gamma_R$ CARPROD[R,S,S]}	γ	SYMMETRICAL
R[$\alpha\leq\beta$]S	γ	$\gamma\leftarrow S[\mathcal{A}]$ *if* $d_R\in\mathcal{A}$ *then* DIVIDE[R,α,$\mathcal{A}$,T,90] *else* DIVIDE[R,α,$\mathcal{A}$,T,a($\mathcal{A}$)]	γ	$\gamma\leftarrow S[\mathcal{A}]$ *if* $\gamma_R\cap\mathcal{A}\neq\emptyset$ *then* {L←$\gamma_R\cap\mathcal{A}$; DIVIDE[R,α,$\mathcal{A}$,T,90]} *else* DIVIDE[R,α,$\mathcal{A}$,T,a($\mathcal{A}$)]

IV SYSTEM DEVELOPMENT

This chapter details the software development of the database management system. Discussion covers the initial system design and anticipated modifications thereto, implementation techniques at the data entry level and conceptual levels, and special utilization of UCSD Pascal in the overall system structure.

SYSTEM DESIGN

Initial Development

Once it was decided to use a relational algebra based DML and DDL, much of the system design followed in a straightforward fashion. Capabilities had to be provided at both the data entry level for inputting and verifying commands, and at the conceptual level for interpretation and global optimization of specific queries. A major breakdown of the system modules is shown in Figure 9. An executive module controls access to all other parts of the system, procedures being initiated from and returning control to it. The exec operates in one of two modes: normal mode under user control, and special mode under control of the database manager (DBM). Special mode is entered by inputting a unique identification password as part of the logon system. Special mode causes the exec and its modules to ignore security restrictions on all relations, thus allowing the DBM to control and maintain the entire database. In addition, priviledged operations such as an initialization command may be used only by the DBM acting in a special mode.

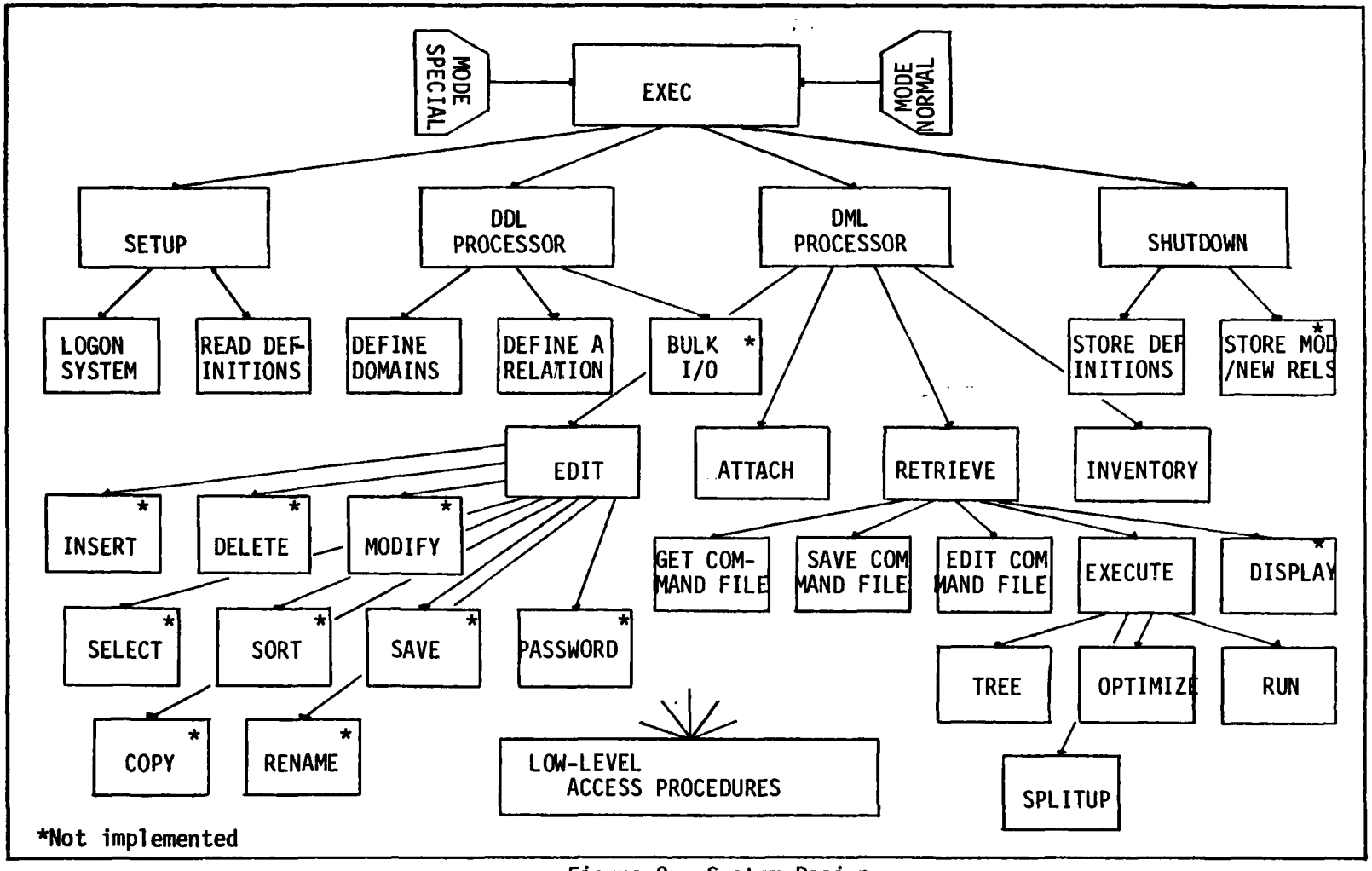


Figure 9. System Design

The DDL processor creates and maintains the domain and relation definitions which exist in the database. These definitions are kept in main memory for quick access by the other modules, and are stored on a disk file when the user quits the system. Since secondary storage consists of removable diskettes, each user may maintain a separate set, with possibly different relations and domains stored on them. The definitions for these are read in during setup time, thereby allowing each user to act as database manager for his or her own database. This feature will also be useful in segmenting unrelated or very large databases. For instance, information about classes, students, and instructors may make up one database, while information about staff and administrative personnel may make up another.

The DML processor controls the execution of all other commands. Four modules, ATTACH, INVENTORY, EDIT, and RETRIEVE comprise the major breakdown of this processor. The purpose of ATTACH is to make relations accessible to the user. This includes setting up any necessary or desired access paths for those relations and checking security rights for later manipulation of those relations. INVENTORY provides the user with a list of the domains and relations he has defined or attached during the current execution of the system.

The EDIT module's function is to execute a variety of utility commands available to the user. These include the ability to insert, delete, and modify tuples in relations, to copy the contents of one relation to another, to select a subset of a relation using any of several functions, renaming relations, sorting relations, and changing the security passwords on relations.

The RETRIEVE module handles the creation and execution of relational queries. Queries are treated by the system as command files which may be changed, stored or retrieved from storage, or executed. Modules exist in RETRIEVE to perform these functions and in addition RETRIEVE also processes commands to display the contents of relations. Further breakdown of the RETRIEVE modules will be described in later sections of this chapter.

Modules also exist to properly shut down the system. New domain and relation definitions must be stored on the disk as well as any relations not already saved or permanently changed.

Separate from all of the above modules are those that handle data at the storage representation level. This is a natural division in a relational database system because the relational view of data assumes nothing about how it is stored. Thus, when better storage models are devised or new hardware is introduced, only these modules need be changed.

Future Modifications

Due to the limited scope of the thesis, Figure 9 shows several modules which have not yet been implemented. Most notably the modules which comprise the storage representation level of operation exist only in theory (See Chapter III). Thus, in order to have some basis for running and testing the other modules of the system, a temporary structure containing the domain and relation definitions was created. The structures used are simple linked lists, one sorted by domain name and the other sorted by relation name.

Since one of the primary advantages of relational systems is its simplicity of data structure description, there is no need to employ different data structures at the conceptual level. Thus when an appropriate set of low-level access procedures are implemented the temporary linked list structures employed above will give way to the same relational model. For each user database there will exist a relation of domain definitions and a relation of relation definitions, and the same low-level procedures which access the database relations may be used to access these "definition" relations.

Lacking a set of low-level procedures, several other modules as shown in Figure 9 were not implemented. Most of these modules have functions which have no applicability if only the definitions of relations exist; i.e., no actual data in the relations. These modules are straightforward in their function and should pose no problem in their future design and implementation. (More detail on the function of these modules can be found in the user's guide, Appendix B).

IMPLEMENTATION TECHNIQUES AT THE DATA ENTRY LEVEL

Computer systems in general are of little use if an appropriate interface to the human user does not exist. Such an interface must be in harmony with both user skills and task requirements. This is especially true in the present case, for this database system is being specifically designed for pedagogical uses in the training of students.

Some of these considerations went into the selection of the relational algebra as the basis for the interface DML and DDL. In addition, consideration of actual machine/user interaction was mandatory. Solutions to two problems were attempted: the handling of over two dozen user commands and the problem of input error detection and/or correction.

The Abundance of Commands

Due to the wide abundance of commands which must be available to the user in a comprehensive database system, a multi-level command system was designed. The hierarchy of commands is shown in Figure 10. At each nonterminal level the user is given a prompt line with a list of commands from which to choose. By entering one character (usually the first character of the command) the user either descends a level into the hierarchy where more choices are available, causes invocation of a procedure or procedures, or quits the current level and returns to the previous level. For example, if a user wishes to change a previously created command file he will type R at the SYSTEM level and a new command line at the RETRIEVE level is displayed. Then by selecting the GET option by typing G the user is able to get his command file from disk into a workfile in memory. Then by typing E (still at the RETRIEVE level) the first 20 or fewer lines of his file is displayed and a new command line at the EDITCOM level is displayed. After using the options at this level to modify his file, he types Q to return to the RETRIEVE level. Now he may save the new version of his file, execute it, etc., or he may simply return to the SYSTEM level by typing Q once again.

The number of options available at each level was limited to between four and six (not counting the Q(uit) option). Fewer options at each level would have meant more levels. While the system would know how deep it was, the user would soon lose track of what levels he came from. On the other hand, if there were more options at each level, the user would get lost among the various options, especially if many commands started with the same character.

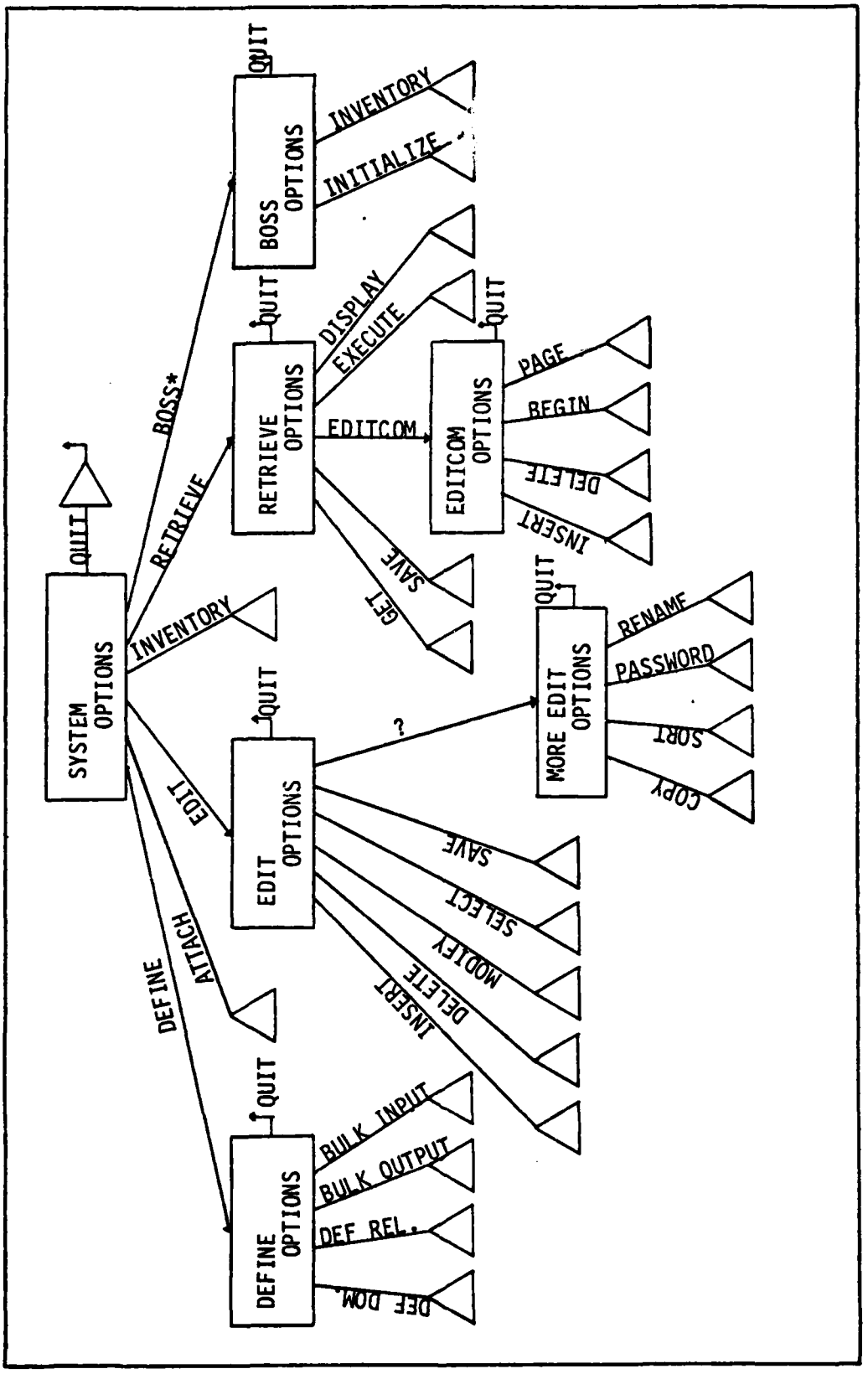


Figure 10. Multi-Level User Command System

Another method of reducing complexity was used in the selection of which commands to make available at each level. Grouping related commands under one option reduces the need to jump from one level to another. For example, after X(ecuting) a command file the user can immediately D(isplay) the result without quitting the RETRIEVE level and perhaps descending another branch at the SYSTEM level. Also, the more commonly used commands which would more likely be called are at a higher level. For example, the options under EDIT are divided into two sets. Those anticipated to be used most often are at the first level of EDIT and any others are at the second level, obtained by typing "?".

Error Detection/Correction

Errors at the Data Entry Level are usually one of two types: (1) the user enters a command and either misspells it, includes an invalid parameter, or specifies an illegal operation; or (2) the user enters a valid command he did not mean to. The first type of error is usually termed a syntax error. These errors are easily detected, but depending on the sophistication of the system, perhaps not as easily corrected. Many times the error can simply be ignored. For example, when an illegal option is entered after a command line is displayed, the command line is simply redisplayed with no action taking place. In other cases the error must be pointed out and either the user allowed to correct his mistake on the spot or as in the case of a command file, ignore the command in which the error occurred and continue to process other commands. In either case, the user should be allowed to correct his mistake or to discontinue execution of the command or command file.

The second type of error is not so easily detected. However, certain precautions can be taken by the system to allow the user to undo these types of mistakes. For instance, consider a user in the RETRIEVE module where he has created or changed a command file. He executes the file and now wishes to leave RETRIEVE and exit the system. However, through oversight he has not saved his new command file on secondary storage -- an easy error, especially if execution took a long time. The system, though, will not allow him to leave the RETRIEVE module without answering a question to throwaway his current workfile with a "Y". This backup method is applied in other instances where it is appropriate.

Some user actions, such as defining a relation which was not wanted or copying one relation to another by mistake may be rectified by simply deleting the relation definition and deleting the copied tuples. However, some operations such as a multiple tuple modification may be impossible or extremely time consuming to back out of if the user does not have a copy of the original relation. Thus all relations created or modified do not take their predecessor's place until the user asks that they be specifically saved as such. Thus, one method of providing permanent backup is to modify the relation, rename it and save it. When the user is sure he no longer needs the original relation it can then be deleted.

IMPLEMENTATION TECHNIQUES AT THE CONCEPTUAL LEVEL

The main thrust of this thesis occurs here in the design and implementation of new techniques at the conceptual level. Only recently has work begun in the optimization of performance at the conceptual level. (See the last section of Chapter 3). Since, with a relational view,

a user query is expressed at such a high level of abstraction, the system itself has the power to make implementation decisions. The presumption is that a "smart" interface can perform better than a nontechnical user who is overwhelmed by a mass of detail. On the other hand, much work remains to be done on how to design the conceptual level interface to efficiently implement a query, let alone provide it with enough intelligence to out-perform a confused or less than proficient user.

The design chosen for this implementation is pictured in Figure 11. This design is an expansion of the basic query optimizer as shown in Figure 8, where TREE corresponds to Syntax Analysis, OPTIMIZE to Tree Transformer, and RUN to Coordinating Operator Constructor. The next four sections describe the algorithms and data structures used in each of the submodules of EXECUTE.

The TREE Module

The TREE module receives as input a pointer to a command file and returns as output a pointer to a network of shared trees. The following algorithm is used for performing this transformation:

For each command in the command file do:
(1) If all elements of the command are present then continue else perform error subroutine.
(2) If all relations to be operated on have been defined and attached and the result relation name is not already in use then continue else perform error subroutine
(3) Create the node and link it to any nodes it uses as operands.

Error subroutine:
(1) Print error message.
(2) Mark command file to indicate error found.
(3) If user wishes to continue having the syntax of his file checked then ignore this command and continue with next one else quit TREE procedure.

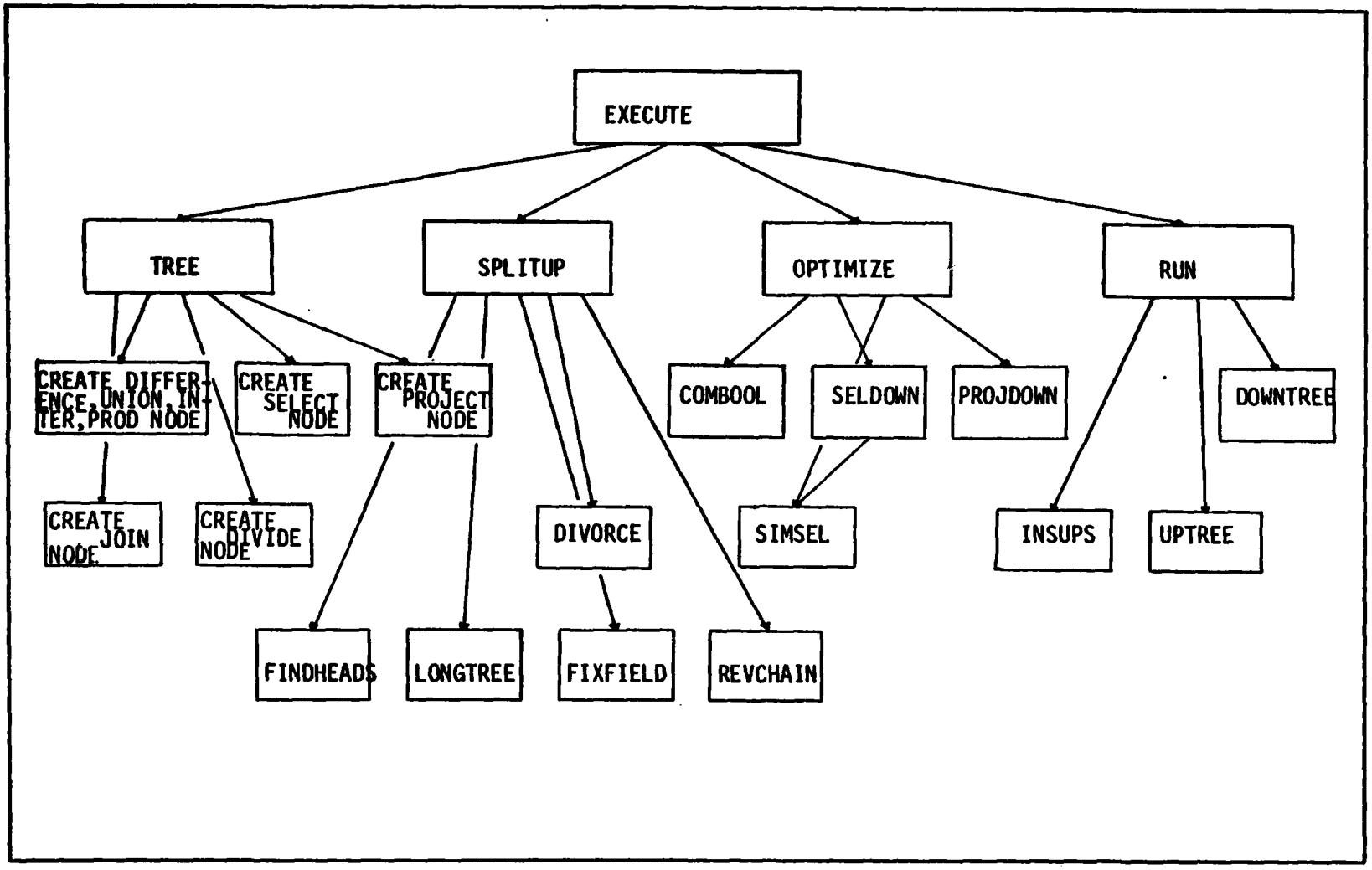


Figure 11. Detail of EXECUTE Module

The data structure used for each node is a record with the following fields:

LEVNUM -- (integer) number of nodes which point to this node.

NODNUM -- (integer) copy of LEVNUM but used later as a counter.

LEFTPTR -- (pointer to node) points to the node which is the left operand or is nil if the left operand is a stored relation.

RIGHTPTR --same as LEFTPTR but refers to right not left.

NEXTNODE -- (pointer to node) used initially to link all nodes together and later to link the root nodes together.

DOWNPTR -- (pointer to node) points to the next node in a pre-order traversal of the tree; used only in RUN.

UPPTR -- same as DOWNPTR but refers to previous not next node.

LEFTNAM -- (string) contains the name of the node pointed to by LEFTPTR or the name of the stored relation if LEFTPTR nil.

RIGHTNAM -- same as LEFTNAM but refers to RIGHTPTR not LEFTPTR.

(Note: if the node is a unary operator such as SELECT then LEFTNAM and/or LEFTPTR are always used.)

RESULTNAM -- (string) the name of this node.

OPERATOR -- (string) the function of this node; e.g., UNION, etc.

PSORESULT -- (pointer to list) points to a list of the preferred sort orders for this node; used only in RUN module.

FIELDRESULT -- (pointer to list) points to a list of the attributes of the relation produced by this node's operation.

VARS1 -- (pointer to list) points to a list containing a boolean expression in postfix notation if this is a SELECT node, to the divide attribute if this is a DIVIDE node, to the join attribute from the left node if this is a JOIN node, to a list of project attributes if this is a PROJECT node, or nil otherwise.

VARS2 -- (pointer to list) points to the join attribute from the right node if this is a JOIN node, or nil otherwise.

HDFLAG -- (boolean) TRUE if this node is a root node of some tree, FALSE otherwise.

STFLAG -- (boolean) TRUE if this node is pointed to by more than one other node, FALSE otherwise.

It is at this point that the differences between the basic query optimizer and the methods used in this thesis become evident. Previous attempts at optimization have considered only single expressions. That is the user formulates a single query and expects a single relation as the result. This thesis has expanded this viewpoint to include multiple queries for which the user expects several relations as the result. Thus the opportunity exists for simultaneous optimization of a set of queries. These opportunities occur in two areas: in the exploitation of shared subtrees not only within a query but also among different queries, and in the execution order of the various queries. These ideas are embodied in the module SPLITUP.

The SPLITUP Module

The SPLITUP module receives as input the pointer to the network of shared trees provided by TREE, and produces as output an order optimized forest of separate trees in which all shared subtrees have been removed. The algorithm for performing this transformation is as follows:

(1) FINDHEADS -- all root nodes in the network are found and linked together using the NEXTNODE field.

(2) While there are root nodes not yet considered by this step do:
    (2.1) LONGTREE -- determine which of the remaining trees has the most number of shared subtrees.
    (2.2) Place the root node for this tree at the beginning of the chain of root nodes.
    (2.3) Continue with remaining nodes.

(3) For each tree, FIXFIELD -- set up the attribute list to be produced by each node in this tree using FIELDRESULT to point to it.

(4) Perform various error checks where applicable:
    (4.1) Ensure relations are union compatible.
    (4.2) Ensure that whenever attributes are referenced, such as in a SELECT predicate, or a PROJECT list, that each corresponds to some attribute in either the FIELDRESULT list of a left or right son or the attribute list of the proper stored relation.
    (4.3) Ensure relations are divide compatible.

(5) DIVORCE -- For each tree do a preorder traversal of the tree where at each node if the leftson/rightson's STFLAG is TRUE and NODNUM count greater than 1 do:
    (5.1) Reduce the leftson/rightson's NODNUM count by 1.
    (5.2) Make the LEFTPTR/RIGHTPTR nil so that when this node is executed a stored relation will be used.

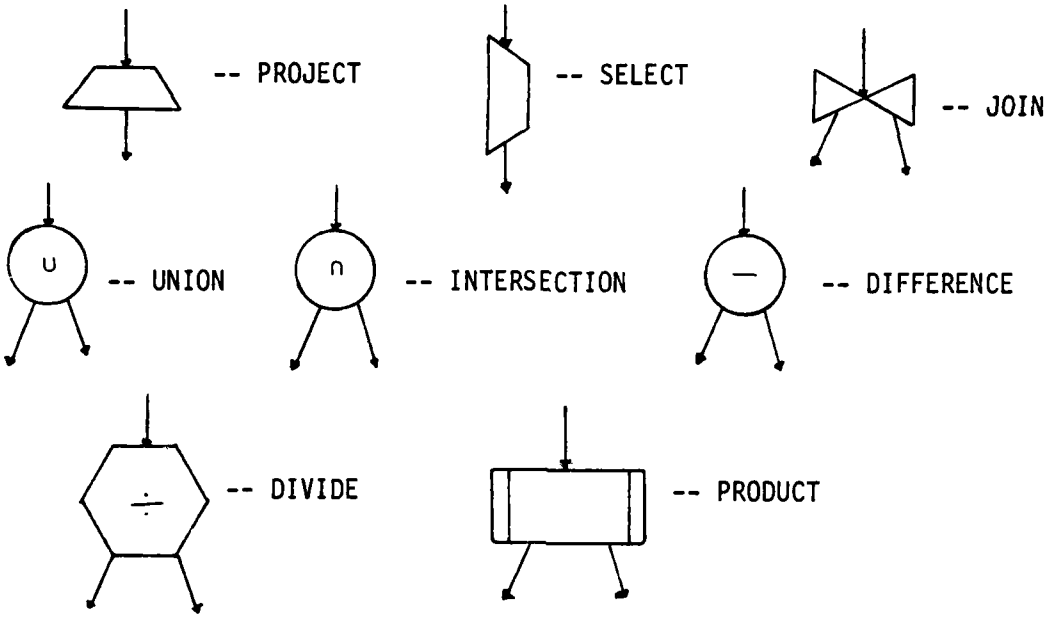
(6) REVCHAIN -- reverse the list of root nodes so that the trees which have the formerly shared subtrees are executed before the stored results of those subtrees are needed.

The importance of step 6 in the above algorithm is obvious. If performance is to be improved by executing shared subtrees only once, it must be ensured that they indeed are executed before the resulting relation is used. However, step 6 only ensures that shared subtrees in different queries are executed properly. Shared subtrees can also occur within the same query and are just as easily eliminated by the SPLITUP algorithm. In order to ensure that these shared subtrees are executed before being used, note that in the elimination step (5) a preorder traversal of each tree was done. Therefore, by executing the nodes of each tree in a reverse preorder, shared subtrees will always be executed before being used.

The OPTIMIZE Module

Once individual operator trees are available, the correctness preserving transformations introduced in Chapter III can be applied to them with the goal of optimizing the performance of the overall query.

Notation: For examples, the parts-supplier model will be used as the stored relations (Figure 1(c)) and the following pictorial notation for the nodes in a tree will be used:



In addition a shorthand notation for expressing queries in a single line is:

A ∪ B -- set union of relations A and B

A - B -- set difference

A ∩ B -- set intersection

A %. T --projection of relation A onto the components given in the projection list T

A : F -- selection of a subset of relation A of tuples for which filter F is true, where F is a boolean predicate involving the components of the tuples of A

A * B -- cartesian product or join on relations A and B

A $\dot{\div}$ B -- division of A and B, where A is a binary relation and B is a unary relation

Two types of tree transformation and one boolean simplification procedure are implemented in the OPTIMIZE module. One type moves unary operators down trees, while the other involves replacing a subtree of set operations on the selections of the same relation by a compound boolean expression. This second type will be discussed first.

The COMBOOL Algorithm. Consider the operator tree shown in Figure 12(a). The relation R may be either a stored relation or an intermediate temporary relation. A direct implementation of this tree would result in R being read three separate times, and several new temporary relations being created. This type of tree will be referred to as a "boolean" tree. One reason for consideration of boolean trees is that a boolean tree is the only way of representing compound boolean restrictions on a relation using the strictest form of Codd's relational algebra. But even if this strict form is not enforced the user may still create queries which result in boolean trees; especially if the predicate would be difficult to understand and formulate if not broken down.

The approach used is to translate boolean subtrees into a single operation having a compound boolean predicate. If R is the common relation, T1 and T2 are trees, and E is any boolean predicate, the translation function is given by:

$\tau\{T1 \cap T2\} = \tau\{T1\}$ and $\tau\{T2\}$
$\tau\{T1 \cup T2\} = \tau\{T1\}$ or $\tau\{T2\}$
$\tau\{T1[E]\} = E$ and $\tau\{T1\}$
$\tau\{R\} =$ true

For example, the tree in Figure 12(a) translates into the compound boolean operation shown in Figure 12(b).

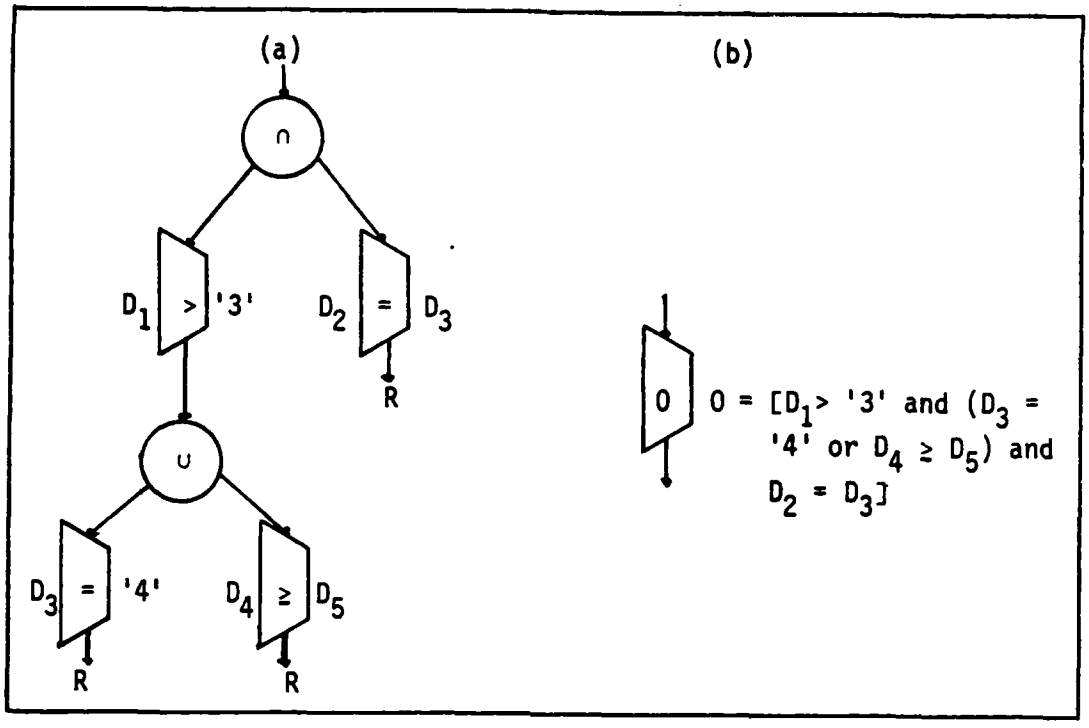


Figure 12. (a) Set operations on SELECTs of the same relation R.
(b) Transformation of the tree in (a) into a single SELECT.

The transformation of boolean subtrees insures that the common relation is never read more than once. Further, if the common relation is stored and has directories available then it may be possible to significantly reduce the number of secondary storage page accessed by the directory analysis.

This transformation can be easily implemented by a recursive procedure where the union or intersection nodes are tested for possible combination. As long as the recursion processes the nodes such that the sons are done before the father, as in postorder, then only one pass is needed to make all possible combinations.

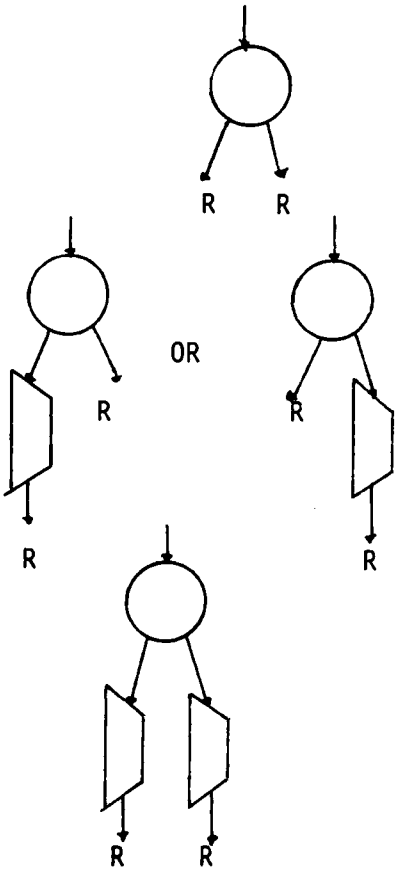
The COMBOOL algorithm implements the above procedure and simultaneously applies the idempotency laws of relational algebra as shown in

Table I (Page 38). Simultaneous application of these procedures avoids the introduction of redundant expressions into the compound boolean predicates and provides optimal simplification (at this level) of the UNION, INTERSECT, and DIFFERENCE operators.

The COMBOOL algorithm:

For each node in a postorder traversal of the tree which is a UNION, INTERSECT, or DIFFERENCE operator do:

CASE "subtree" of:



: apply law 1,2 or 3 from Table I whichever is appropriate.

: apply law 4,5 or 6 from Table I whichever is appropriate.

: combine into single SELECT operation.

The SIMSEL Algorithm. SIMSEL is an algorithm for simplifying the boolean predicates of SELECT operations. In addition, each predicate is put into a standard form which is necessary for the SELDOWN algorithm (see later section) and which will also make implementation easier and

more efficient. The TREE procedures take the boolean expression entered by the user and put it into postfix form. The purpose of SIMSEL is to take this expression, put it into conjunctive normal form and simplify this result as completely as possible.

The SIMSEL algorithm:

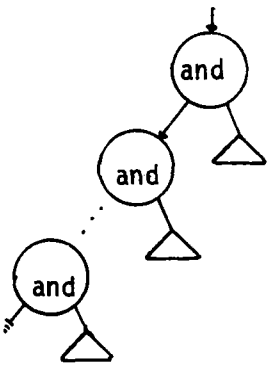
For each SELECT node in the tree do:

(1) Reverse the postfix order of the expression.

(2) Create an expression tree using an inorder recursive procedure accessing sequentially the result of (1).

(3) Distribute OR operators into AND operators:
$(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$.
$(x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$.

(4) Rotate the tree so that all AND operators are moved left as far as possible. At this point the tree looks like:



This is now in the standard conjunctive normal form. Each triangle is called an orgroup if it contains at least one OR operator or it is an atomic node.

(5) Transform each orgroup by eliminating all redundant nodes. This is expressed by the identities adapted from Table II, where X is an orgroup, y is an atomic node:

$y \vee y = y$
$y \vee (y \vee X) = (y \vee X) \vee y = y \vee X$
$y \vee (X \vee y) = (X \vee y) \vee y = X \vee y$

(6) Transform each pair of orgroups using the following boolean identities, where X, Y are orgroups, y is an atomic node:

$X \wedge X = X$
$y \wedge y = y$
$X \wedge y = y \wedge X = y$, if y is in X.
$X \wedge Y = Y \wedge X = Y$, if all nodes in Y are in X.

(7) Recreate the expression in postfix order.

The SELDOWN and PROJDOWN Algorithms. The other type of tree transformation moves the unary operators, SELECT and PROJECT, down the operator tree. These are implemented via the SELDOWN and PROJDOWN algorithms, respectively..

The SELDOWN Algorithm. Consider the two relations SUPPLIER and SP. The query "Find the name and status of all suppliers who supply any part in the quantity greater than 200" can be expressed by a user in the relational algebra as:

```
JOIN SUPPLIER, SP WHERE S# = S# GIVING T1
SELECT ALL FROM T1 WHERE QTY   200 GIVING T2
PROJECT T2 OVER SNAME, STATUS GIVING T3.
```

T1, T2 and T3 are temporary relations formed by the operations of JOIN, SELECT, and PROJECT respectively. This query, represented as an operator tree is shown in Figure 13(a).

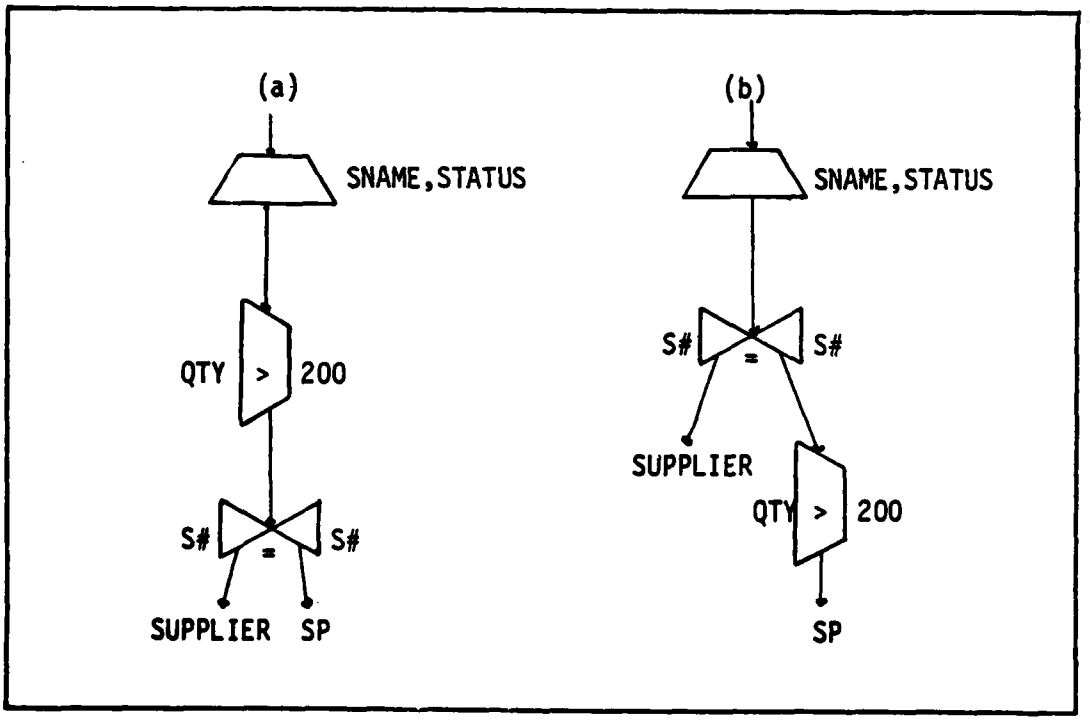


Figure 13. (a) The tree for a user query, (b) The tree for a transformation of the query in (a).

At this stage the tasks in the query are clearly identified as the operations of JOIN, SELECT, and PROJECT. Without considering the detailed implementation of SUPPLIER, SP, and these operations, a good programmer would know that there are correctness-preserving transformations which can be applied to improve efficiency at this level of abstraction. In particular it is beneficial to move SELECT operations as far down the tree as possible using such transformations. This is because the SELECT operation reduces the number of tuples to be processed by subseuqent operations. Any reduction is particularly advantageous when JOIN (or PRODUCT) operations occur later. In general a JOIN operation on two relations, A and B, has to be performed as a full Cartesian product to produce n(A)n(B) tuples from which the relevant tuples are selected. However, any part of the selection filter which is moved through the join is executed n(A) times rather than n(A)n(B) times (or vice-versa). In the present case, the SELECT operation rejects those joined tuples whose QTY is not greater than 200. The effort involved in joining tuples which are subsequently thrown away is wasted. Since the QTY domain occurs only in the SP relation, selection can be performed on SP before joining the result with SUPPLIER. After applying this transformation the tree appears in Figure 13(b).

Moving SELECT operations down the tree is straightforward in most cases. As Table III shows, in all cases except JOIN and PRODUCT, the select operation is either "distributed" through the other operation or in the case of a sequence of SELECTs the filters are simply concatenated. Some alternatives are given that require less filtering, but these are not preferred for two reasons. First, the reason for the transformations is that filters reduce cardinality significantly and so

should be worth the repetition. Second, by making a symetric distribution, common subexpressions that lead to successful application of the idempotency laws discussed in the next to last section are not destroyed.

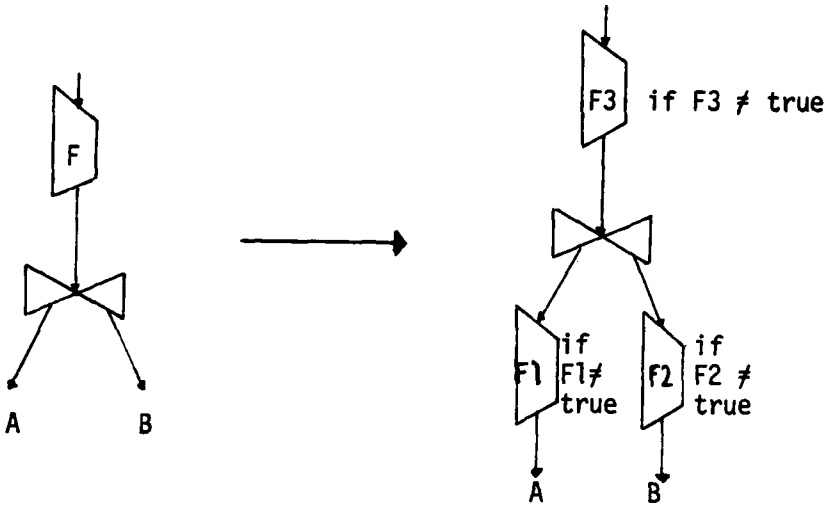
The only really complex transformation occurs for joins. In general the filter (F) associated with a join refers to both of the relations (say A and B) being joined. However, there may be parts of it that refer only to one or the other of the argument relations, and this part could possible be factored out and moved down through the join. Thus, it is desired to transform filter F into an equivalent F1 and F2 and F3, where filter F1 refers only to components of relation A, filter F2 refers only to components of relation B, whereas filter F3 refers to both relations A and B. Clearly F1 and F2 should be as large as possible. The algorithm for doing this is:

(1) Apply the boolean simplification procedure of SIMSEL to the SELECT node. Now F will be in conjunctive normal form.

(2) Set F1 = F2 = F3 = true.

(3) For each conjunct x of F do:
```
If all elements of the conjunct refer to relation A then
  F1 = F1 ∧ x
else if all elements refer to relation B then
  F2 = F2 ∧ x
else
  F3 = F3 ∧ x.
```

(4) Transform tree:



In order that each SELECT operation is moved down the tree as far as possible the following general algorithm is used:

SELDOWN Algorithm: For each node in a preorder traversal of the operator tree which is a SELECT operation do: apply the appropriate distribution law from Table III.

Notes on the Efficiency of Moving SELECTs. Transformations like the distribution of a filter into a UNION do not necessarily improve things. If A and B are disjoint, then the cardinality of their union is the sum of their individual cardinalities and filter F is applied as many times both with and without the transformation. Thus applying the distribution must always be favorable. But if A and B overlap, then for some tuples, if the filter is distributed, it will be applied twice, which may not compensate for the saving of work in the union obtained by performing the filtering first. In some cases, the filter may not change the cardinality much, and then there is a loss by moving the filter down the expression tree. For intersection and difference this worsening of the situation when the filter does not change the

cardinality significantly is even more marked. However, as noted before, for joins -- especially where the join is necessarily a full quadratic join -- the transformation almost always improves things.

Note, however, that moving SELECTs down to the leaves of the operator tree has an important added advantage. During the evaluation, the relation at a leaf is stored, and the presence of a filter would enable the system to use any indexes or inversions present. This could lead to significant savings compared to the alternative of reading through all the tuples of the relation and selecting those desired using the filter. This latter course is the only course available for filters positioned at nodes other than leaves.

The PROJDOWN Algorithm. There are also benefits to be gained by moving PROJECT operations down a query tree. PROJECT operations decrease the width of tuples and, due to the elimination of duplicate tuples, may also decrease the number of tuples in a relation. In Figure 13(b) the PROJECT operation retains only the SUPPLIER attributes SNAME and STATUS. It is therefore sufficient to supply JOIN with only the SNAME, STATUS and S# attributes from SUPPLIER, and the S# attribute from the SELECT operation. Figure 14(a) shows how the tree appears when additional PROJECT operations are included before JOIN.

Notice that the original PROJECT must be retained in order to eliminate the S# attribute before output. Each of the new PROJECT operations can be retained or removed individually depending on whether or not it increases efficiency. This decision is influenced by the fact that the implementation of PROJECT operations cannot take advantage of directories, whereas the implementation of JOIN often can.

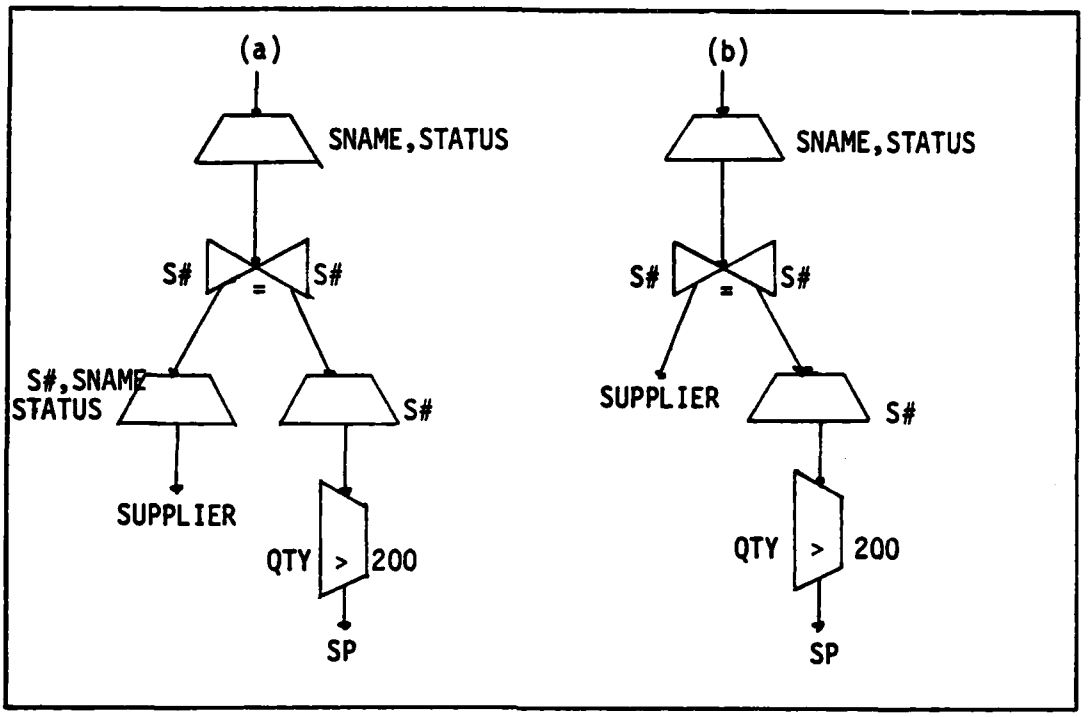


Figure 14. (a) Further transformations on the tree of Figure 13(b). (b) Final optimized tree for the query of Figure 13(a).

The PROJECT operation over SUPPLIER will shield JOIN from using any existing directories. The PROJECT is therefore deleted. The other PROJECT is retained, since there are no directories associated with the output of the SELECT operation. The final optimized tree is shown in Figure 14(b).

In contrast to the SELECT operation, the distribution laws for PROJECT are somewhat more complex, in that several of the laws are only applied when certain criteria hold. These laws are given in Table IV. For the same reasons SELECT operations were moved down to the leaves of the tree, it is desired not to move certain PROJECT operations there. For instance, implementations of INTERSECT and DIFFERENCE need only compare primary key attributes to determine tuple eligibility for the result relation and especially at a leaf node there generally

exist directories on these keys. Therefore, PROJECT operations which would remove these keys should not be distributed through the particular operation . Care must also be taken in moving a PROJECT operation through a SELECT, in that by performing the PROJECT first, attributes referred to by the boolean expression in the SELECT operation are not eliminated.

Once again the only complex transformation occurs for joins. The projection list (T) can be broken into two disjoint sets, one containing attributes from the first joined relation and the others from the second joined relation. Thus, in general the PROJECT operation can be split into two PROJECTs and moved through the JOIN. However, as brought out in the example, moving a PROJECT through a JOIN at a leaf shields the JOIN from the use of directories and thus lowers efficiency. In addition, the original PROJECT must be retained if it does not include both of the attributes used in the JOIN condition, or if the PROJECT is not moved down one or both of the branches of the JOIN. (Note in a PRODUCT operation it is never necessary to keep the original PROJECT.)

Two further reductions can be accomplished when moving PROJECTs. First, if one of the projection lists created as a result of moving a PROJECT through a JOIN is empty, then the JOIN is a useless operation and can be eliminated, along with the branch for which the projection was empty. Second, if a PROJECT operation does nothing; i.e., the projection list contains all of the attributes of the input relation, it can be eliminated. The algorithm for PROJDOWN is as follows:

PROJDOWN Algorithm: For each node in a preorder traversal of the operator tree which is a PROJECT operation do:

(1) Eliminate it if possible.

(2) Apply the appropriate distribution law from Table IV, eliminating a JOIN if possible.

Modification of the Transformation Algorithms. The transformation algorithms given above have one fatal flaw. As they stand, the possibility exists for destroying the shared subtrees found with the SPLITUP procedures. Thus, a simple test to see if a shared subtree would be violated should be inserted in each algorithm. Each shared subtree can be optimized individually and since nodes are never moved up the tree, the recursive algorithms above will automatically do this. The only side effect occurs when in the PROJDOWN algorithm, a JOIN operation and one of its branches can be eliminated. If any or all of the branch to be eliminated is a shared subtree then a way of executing that branch must be implemented. The problem was solved by noting that the JOIN is only eliminated as a result of trying to move a PROJECT through it. Since PROJECT is a unary operation, the shared subtree is attached to the PROJECT as a second argument. This will ensure that the subtree is executed without affecting the actual operation of the rest of the tree.

The OPTIMIZE Algorithm. Now that the various optimization algorithms have been described, the overall OPTIMIZE algorithm can be specified. The order of application of the various algorithms is quite flexible; however, some are better than others. Smith and Chang (Ref 12) and Hall (Ref 14) both suggest moving select operations down the tree as their first step. Smith and Chang further recommend that combining boolean subtrees be the next step and then another application of moving the now combined selects down the tree. In terms of the algorithms described here the order would be:

1. SELDOWN
2. COMBOOL
3. SELDOWN
4. or 5. SIMSEL or PROJDOWN.

This order is inefficient in that the SELDOWN procedure must be called twice, and in that within that procedure SELECT operations are unnecessarily moved through boolean subtrees which are to be combined later. Thus this implementation uses the following OPTIMIZE algorithm:

Without violating shared subtrees do:

(1) COMBOOL -- combine boolean subtrees on a common relation

(2) SELDOWN -- move SELECT operations as far down the tree as possible; calling SIMSEL's simplification procedure when moving through a JOIN

(3) SIMSEL -- simplify and standardize the boolean predicates of SELECT operations.

(4) PROJDOWN -- move PROJECT operations as far down the tree as possible but never through a SELECT or JOIN operation at a leaf.

The RUN Module

Whether or not the tree is optimized, procedure calls must be generated to perform the operations specified. However, there is a certain amount of optimization which can be done in this regard as well. This part, developed in (Ref 12), is known as the coordinating operator constructor (Refer to Figure 8). The coordinating operator constructor takes an operator tree and implements each operator from a set of basic procedures in such a way that the sort orders of intermediate relations are optimally coordinated. The set of basic procedures is described in Appendix C.

Each procedure is designed on the premise that a relation is always large compared to one of its directories. Each procedure operates on

"piped" relations whenever feasible so as to avoid writing and reading temporary relations to and from secondary storage. However, in several binary operators (e.g., PRODUCT) it is essential to have one operand as a stored relation, otherwise one operand subtree must be evaluated repeatedly. The procedures always access stored relations sequentially so as to avoid the high time overhead associated with random access on secondary storage devices. When a temporary stored relation is used repeatedly by some operator (e.g., JOIN) directories may be first created to speed access and further reduce paging overhead. Each procedure assumes its input relations do not contain duplicates, and UNION and PROJECT remove any duplicates they create before passing the relation up the tree.

SELECT operations are implemented by one generalized procedure. This procedure is essentially an extension of the "list combining" and "test tree" algorithms developed in (Ref 19). The idea is to utilize directories to reduce the number of secondary storage pages which must be accessed. The SELECT procedure always outputs tuples in the same order as they are input.

There are three JOIN procedures. JOIN1 is only used for the equality predicate in a frequent degenerate case, when one relation has a single domain. Tuples are output in an order determined by the other (multi-domained) relation. JOIN2 is used for most cases of the equality predicate. The output relation is sorted on both "joining" domains. JOIN3 is used for nonequality predicates. Tuples are output in an order that corresponds to one of the input relations.

There are four procedures for each of the set operators UNION, INTERSECT, and DIFFERENCE. The four procedures for UNION are typical.

The idea is to exploit any common fast access path (i.e., sort order or directory) between the two input relations. UNION1 exploits a common sort domain to output tuples, without duplicates, sorted on the common domain. UNION2 exploits directories over a common domain. Output tuples are not in any consistent sort order. UNION3 takes advantage of a situation in which one relation is sorted on domain D and the other relation has a directory over D. Tuples are again output in no consistent sort order. UNION4 is only utilized when there is no common fast access path.

Now the scheme for coordinating the selection of sort orders for relations passed between operators can be described. From a consideration of the basic procedures, it follows that there are circumstances in which an operator at some node of the operator tree can be implemented by any one of several procedure calls with comparable efficiency. Further, each of these procedure calls will give a different output sort order. An element of choice is therefore introduced into the sort order of an intermediate relation. Such a choice can often be maintained (or even increased) in intermediate relations on higher branches of the tree. The sort order for an intermediate relation must be chosen so as to lead to an optimum overall implementation of the operator tree.

As mentioned in Chapter III, Smith and Chang have developed a relatively rudimentary, but often highly effective method for coordinating the sort orders of the nodes in a tree, which involves two passes, up and down, through the operator tree. As noted the UP-rules for the upward pass are listed in Table V and the DOWN-rules for the downward pass are listed in Table VI. The rules differ depending on whether the node in question is: (1) at a leaf or (2) completely internal,and, in the case of binary operators, whether a node has (3) a leaf left operand and an

internal right operand or (4) an internal left operand and a leaf right operand. These rules were generated from the set of basic procedures in Appendix C by considering the relative efficiency of each procedure in all relevant situations (for more details see (Ref 20)).

To continue the example, the UP-rules and DOWN-rules will be applied to the optimized tree in Figure 14(b). Assume that the primary key for SUPPLIER is S# and the primary keys for SP are S#, and P#. Figure 15(a) shows the result of applying the UP-rules to this tree in the case that SUPPLIER is sorted on S# and SP is sorted on S# and directories exist for the primary keys.

First the SUPPLIER branch is labeled with {S#}, since it is in this order that tuples will always be retrieved from the SUPPLIER relation. Similarly the SP branch is labeled with {S#}. Note that if SP were not sorted the branch would have been labeled with {n}. The UP-rule for SELECT (R[E]) indicates that an efficient implementation of a SELECT operation at a leaf will preserve the sort order (d_R) of the stored relation. Therefore, the output of SELECT is labeled with {S#}. The UP-rule for PROJECT (R[β]) at an internal node indicates that if there are any common domains between the input pso set (γ_R) and the set of projected domains (β), then it is most efficient to output tuples sorted on one of these common domains. In this case ($\gamma_R \cap \beta$) = {S#}, and the output of PROJECT is so labeled. The UP-rule for JOIN (R[C=D]S), where one operand (R) is at a leaf and the other (S) is internal, indicates that if S is [illegible] and a directory exists for the joining domain (C) of R then the preferred output sort order is d_R. In this case d_R = S# and C = S#, and so [illegible] labeled with {S#}. Finally, for the upper PROJECT, [illegible] empty, and so the UP-rule requires

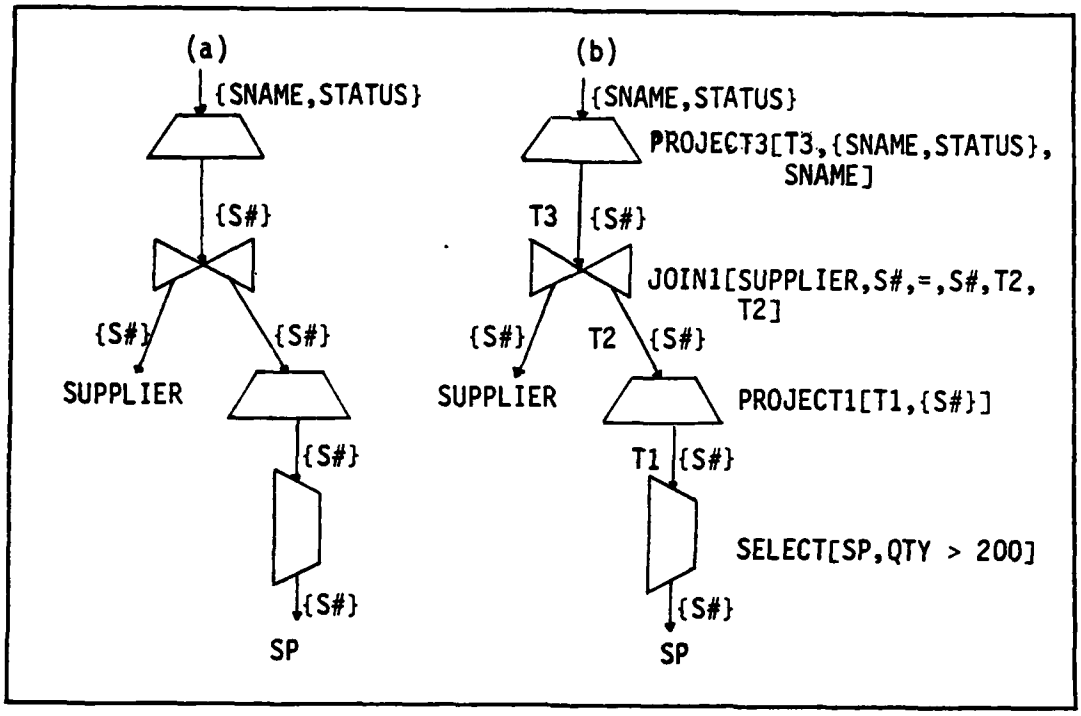


Figure 15.(a) UP-rules applied to the tree in Figure 14(b).
(b) DOWN-rules applied to the tree in (a).

Now the DOWN-rules are applied starting at the root and working down. Since $(\gamma_R \cap \beta)$ is empty, the DOWN-rule for PROJECT specifies that that this node is to be implemented by the PROJECT3 procedure. The third parameter of this procedure, which determines the output sort order, is any arbitrary domain in the output pso set. In this case SNAME is arbitrarily chosen. The DOWN-rule for JOIN states that the JOIN1 procedure should be used to implement this node. Similarly, PROJECT1 and SELECT are specified for the remaining nodes.

The net result is that each operator is implemented as efficiently as possible for the available input sort order(s). The only operation which is denied the use of its most efficient algorithm is the upper PROJECT: its input sort order is such that it must eliminate duplicates by sorting on SNAME.

In this example there were no choices of pso at any branch other than the root. This implies that operators could have been bound to their implementations on the upward pass of the tree. In general there will be a choice and binding must then be done on the downward pass. This is illustrated in the next example.

Figure 16 gives a tree which cannot be further optimized by the use of tree transformations. It is assumed that R1 is sorted on domain A, R2 is sorted on domain D, and R3 is not sorted. Application of the UP-rules gives the labeled tree shown in Figure 17(a). Notice that there are several branches with two or more pso's. The choice in this case is introduced by the two leaf PROJECT operations. Figure 17(b) shows how the DOWN-rules have coordinated sort orders and bound implementation procedures so as to take advantage of this choice. Examine the advantages of the decisions made by the DOWN-rules in this example.

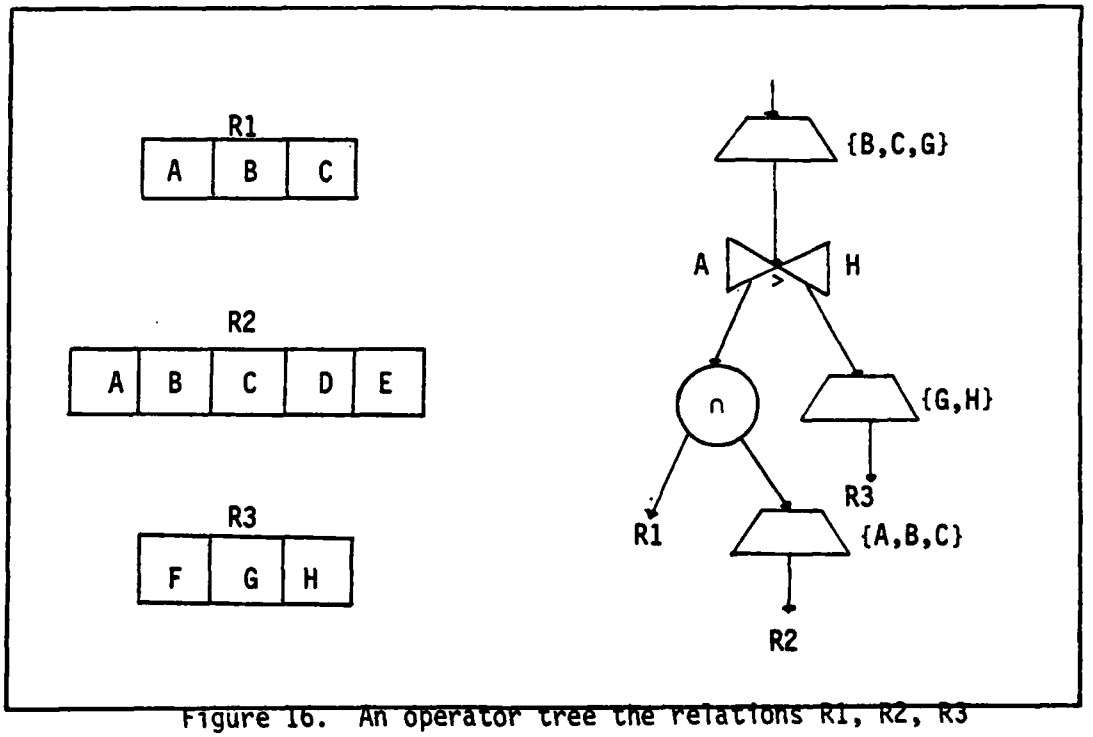


Figure 16. An operator tree the relations R1, R2, R3

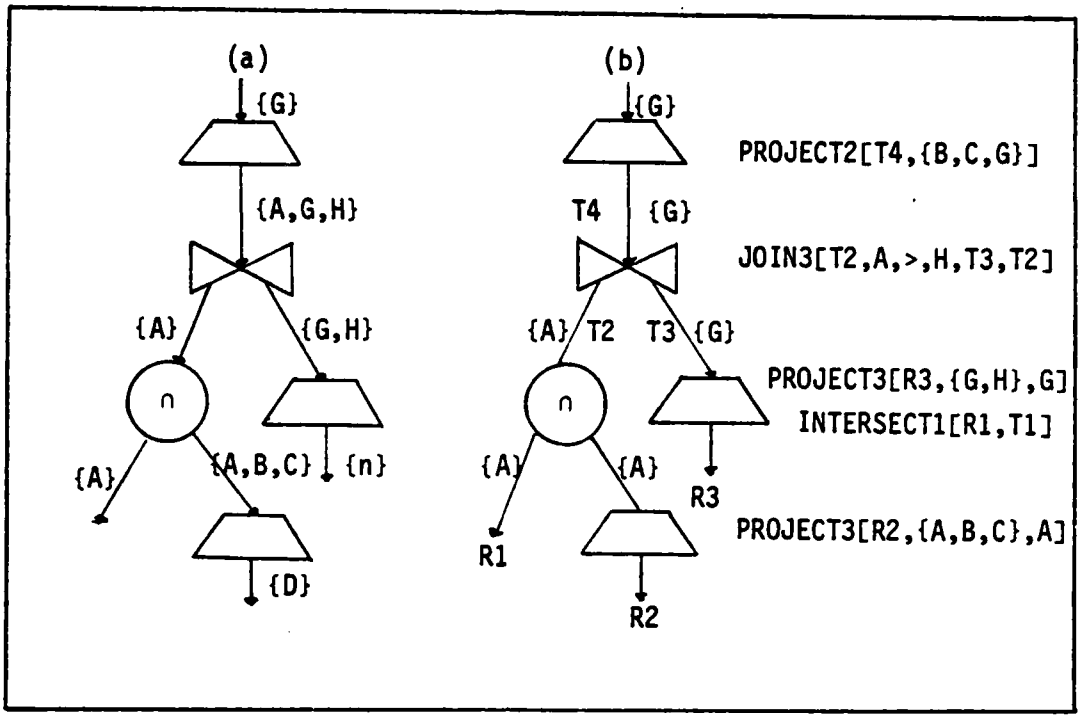


Figure 17. (a) UP-rules applied to the tree in Figure 16.
(b) DOWN-rules applied to the tree in (a).

The advantage of outputting tuples sorted on A from the PROJECT operation on R2 is that both INTERSECT and JOIN can be implemented efficiently. INTERSECT will have both its operands sorted on a common domain and can therefore use a fast "merge" procedure. This intersection procedure automatically outputs tuples sorted on A. In the case of a non-equality comparison, JOIN requires one operand to be stored in sort order on its "joining" domain. Since the "joining" domain of T2 is A and T2 is already sorted on A, JOIN can be implemented without additional sorting.

The advantage of outputting tuples sorted on G from the PROJECT operation on R3 is that the output PROJECT operation does not need to perform additional sorting. Since T2 is stored in the appropriate sort order, JOIN will pipe T3 so as to produce tuples which are also sorted on G. The output PROJECT therefore receives tuples sorted on one of the

projected domains. This PROJECT can then remove duplicates by local comparison rather than by sorting.

From Tables V and VI it can be seen that the UP-rules and DOWN-rules take into consideration only directory and sort order availability. In general this is the only information known about relations at the time when optimization is performed. However, it will be noticed that the function a(Y), which means select an arbitrary element of Y, is utilized in several rules. This function can be implemented by either using a random selector or, if some method of predicting run-time condition exists, selection could, for example, be from the predicted smallest relation, or a domain whose directory is predicted to have the least number of distinct values.

After the DOWN-rules have been applied, some branches may still be labeled with pso sets containing several elements. This implies that the upper operations are insensitive to a particular set of sort orders, and so the sort order can be selected at run-time when more information is available.

The RUN Algorithm. The RUN algorithm incorporates the above procedures with the necessary steps to ensure shared subtrees are executed before being used:

(1) INSUPS -- link nodes of the operator tree in preorder through the DOWNPTR field and in reverse preorder through the UPPTR field.

(2) UPTREE -- accessing nodes via UPPTR, determine the pso set for each node using UP-rules.

(3) DOWNTREE -- accessing nodes via DOWNPTR, create an implementation of each node using DOWN-rules.

(4) Execute the procedure calls created in (3), storing an intermediate result if STFLAG is TRUE; i.e., this is a shared subtree.

Implementation Techniques at the SYSTEM Level

In Chapter II, the hardware and operating system on which the database system was to run were introduced. In this section some of the interfaces to that operating system are discussed.

The current size of the entire database system is approximately 50K bytes. By the time low-level access procedures are implemented the system will be well over the 64K suggested minimum for running the system. However, a very important feature of UCSD Pascal consists of its capabilities for segmentation and overlays. Any program may be broken into a maximum of seven SEGMENT procedures. Thus non-overlapping code can be put into separate SEGMENT procedures where the code and data for each SEGMENT are in memory only while there is an active invocation of that procedure. For example, one-time code, such as the SETUP module, can be put in a SEGMENT procedure. Then after performing the module, the now-useless code is taken out of memory thus increasing the available memory space.

The UCSD Pascal system also supports a facility for integrating externally compiled and assembled routines and data structures. These modules are know as UNITs. More precisely a UNIT is a group of interdependent procedures, functions, and associated data structures which perform a specialized task. Whenever this task is needed within a program, the program indicates it USES the UNIT. One UNIT is used in the database system. It contains data structures and subroutines used by more than one segment procedure. In particular all procedures which perform special functions on the terminal are included here. Thus if the terminal is changed only the procedures in the UNIT need be changed.

V VERIFICATION AND VALIDATION

Verification and validation are important aspects in the design of any new system. They are especially important in this system design because the algorithms used are new in their conception and ever newer in their implementation. Verification of the current system is fairly straightforward. As with most systems, testing is performed until the system must be delivered. However, testing can be structured just as the system is modularized. For instance, the procedure which moves a SELECT operation through a UNION operation can be tested by creating an appropriate tree, passing it to the procedure to be tested, and examining the tree afterward. A procedure exists to print out an operator tree for such examination. Since many of the tree algorithms are recursive, it is best to test operations on them at the root node, an internal node, and a leaf node. Verification is also important at the Data Entry Level, since the system must ensure that only "valid" inputs are accepted and "invalid" inputs are rejected.

The process of validation is somewhat more difficult. Since the system is not complete, there is no feasible way to see if the optimization algorithms actually improve performance. At this point only previous validation studies can be examined. In his article on optimizing single expressions (Ref 14), P. A. V. Hall has provided the results of an experimental validation of a system employing some of the same procedures used in the system described herein. (Notably missing are the algorithms COMBOOL and except for the combining of sequences of projections, the algorithm PROJDOWN.) The reader is referred to Hall's article for details, but the basic results are summarized here.

Hall formulated seven queries from simple to complex and ran them on three levels of his optimizer with each level adding more "optimization". Figure 18 shows the breakdown of the three levels. Table VII describes the relations used in his test and Table VIII lists the queries tested using the notation of Chapter IV. The results are presented in Table IX. Hall concluded from the results that "the transformations do catch the extreme cases without degrading well formulated queries to a great extent. In practice the sizes of relations would be orders of magnitude larger, and the savings would be more significant. The overhead for the optimizer would become completely insignificant."

Table VII Relations Used in Tests of Optimizers (Ref 14)

Name	Relations Degree	Cardinality	Components 1	2	3	4
ACQ	4	1000	acquisition number sequence 1 to 1000	author uniform 1, 10000	title uniform 1, 100000	Dewey Code uniform 1, 200
BRW	4	10	borrower number sequence 1 to 10	name uniform 1, 10000	address uniform 1, 10000	status uniform 1, 10000
DDC	2	200	Dewey code sequence 1 to 2000	subject uniform 1, 10000		
HIST	4	1000	acquisition number uniform 1,1000	borrower number uniform 1,10	date in uniform 1, 10000	date out uniform 1, 10000

Table VIII. Expressions used as queries in testing optimizers. The original expressions are shown together with the expressions to which they are transformed by the three levels of optimizer. (Ref 14)

Test No.	Level	Expression
1	original	HIST-HIST
	Mini	HIST-HIST
	Midi	∅
	Maxi	∅
2	original	((HIST % C_2,C_3,C_1) % C_1,C_3) % C_2
	Mini	HIST % C_1
	Midi	HIST % C_1
	Maxi	HIST % C_1
3	original	(ACQ ∪ (BRW ∪ HIST)): $C_3 < 3$
	Mini	ACQ:$C_3 < 3$ ∪ (BRW:$C_3 < 3$ ∪ HIST:$C_3 < 3$)
	Midi	ACQ:$C_3 < 3$ ∪ (BRW:$C_3 < 3$ ∪ HIST:$C_3 < 3$)
	Maxi	ACQ:$C_3 < 3$ ∪ (BRW:$C_3 < 3$ ∪ HIST:$C_3 < 3$)
4	original	(ACQ ∪ (BRW ∪ HIST)):$C_3 > 2$
	Mini	ACQ:$C_3 > 2$ ∪ (BRW:$C_3 > 2$ ∪ HIST:$C_3 > 2$)
	Midi	ACQ:$C_3 > 2$ ∪ (BRW:$C_3 > 2$ ∪ HIST:$C_3 > 2$)
	Maxi	ACQ:$C_3 > 2$ ∪ (BRW:$C_3 > 2$ ∪ HIST:$C_3 > 2$)
5	original	ACQ * HIST:(C_1=C_5 & $C_2 < C_7$ & $C_3 < 100$ & $C_6 > 8$ &($C_7 < 100$|$C_3 < 100$)
	Mini	ACQ * HIST:(C_1=C_5 & $C_2 < C_7$ & $C_3 < 100$ & $C_6 > 8$ &($C_7 < 100$|$C_3 < 100$)
	Midi	((ACQ:$C_3 < 100$) * (HIST:$8 < C_2$)):C_1=C_5 & $C_2 < C_7$ &($C_7 < 100$|$C_3 < 100$)
	Maxi	HIST:$8 < C_2$ → TEMP ((ACQ:$C_3 < 100$) * TEMP):C_1=C_5 & $C_2 < C_7$ & ($C_7 < 100$ $C_3 < 100$)

(continued)

Table VIII
(continued)

Test No.	Level	Expression
6	original	((ACQ ∪ BRW) ∩ (BRW ∪ HIST)) % C_1 - ((ACQ ∪ BRW) ∩ (BRW ∪ HIST)) % C_2
	Mini	((ACQ ∪ BRW) ∩ (BRW HIST)) % C_1 - ((ACQ ∪ BRW) ∩ (BRW ∪ HIST)) % C_2
	Midi	((ACQ ∪ BRW) ∩ (BRW ∪ HIST)) % C_1 - ((ACQ ∪ BRW) ∩ (BRW ∪ HIST)) % C_2
	Maxi	((ACQ ∪ BRW) ∩ (BRW ∪ HIST)) → TEMP1 TEMP1 % C_2 → TEMP2 TEMP1 % C_1 → TEMP3 TEMP2 - TEMP3
7	original	(((BRW * HIST * ACQ * DDC):C_1=C_6 & C_5=C_9 & C_{12}=C_{13}) % C_1,C_2,C_3,C_4,C_{14}):C_5=3927) % C_1,C_2,C_3,C_4 ∩ (((BRW * HIST * ACQ * DDC):C_1=C_6 & C_5=C_9 & C_{12}=C_{13}) % C_1,C_2,C_3,C_4,C_{14}):C_5=9315) %C_1,C_2,C_3,C_4
	Mini	((BRW * HIST * ACQ * DDC):(C_1=C_6 & C_5=C_9 & C_{12}=C_{13}) & C = 3927) % C_1,C_2,C_3,C_4 ∩ ((BRW * HIST * ACQ * DDC):(C_1=C_6 & C_5=C_9 & C_{12}=C_{13}) & C_{14}=93% C_1,C_2,C_3,C_4
	Midi	(((BRW * HIST:C_1=C_6) * ACQ:C_5=C_9) * (DDC:C_2+3927):C_{12}=C_{13} % C_1,C_2,C_3,C_4 ∩ ((BRW * HIST:C_1=C_6) * ACQ:C_5=C_9) * (DDC:C_2=9315):C_{12}= % C_1,C_2,C_3,C_4
	Maxi	DDC:C_2 = 9315 → TEMP1 DDC:C_2 = 3927 → TEMP2 (BRW * HIST:C_1=C_6) * ACQ:C_5=C_9 → TEMP3 (TEMP3 * TEMP2:C_{12}=C_{13}) % C_1,C_2,C_3,C_4 ∩ (TEMP3 * TEMP1:C_{12}=C_{13}) % C_1,C_2,C_3,C_4

Table IX. Times taken to answer the queries of Table VIII without optimizers and with the three levels of optimizer. Times shown are the smallest of a series of trials (excepting query 7, in which only one trial was made). The measurements were made on an IBM System 370, model 145 using multi-access system CMS, with between 10 and 15 active users during the trials [Ref 14].

	CPU time taken (s)			
Query no.	no-opt	Mini	Midi	Maxi
1	2.01	2.21	0.73	0.81
2	10.37	1.20	1.69	2.22
3	3.78	2.73	2.74	2.95
4	5.49	5.73	5.83	8.41
5	4.13	4.36	3.60	3.76
6	3.87	4.12	4.25	3.23
7	estimated 10 days	estimated 10 days	1401	693

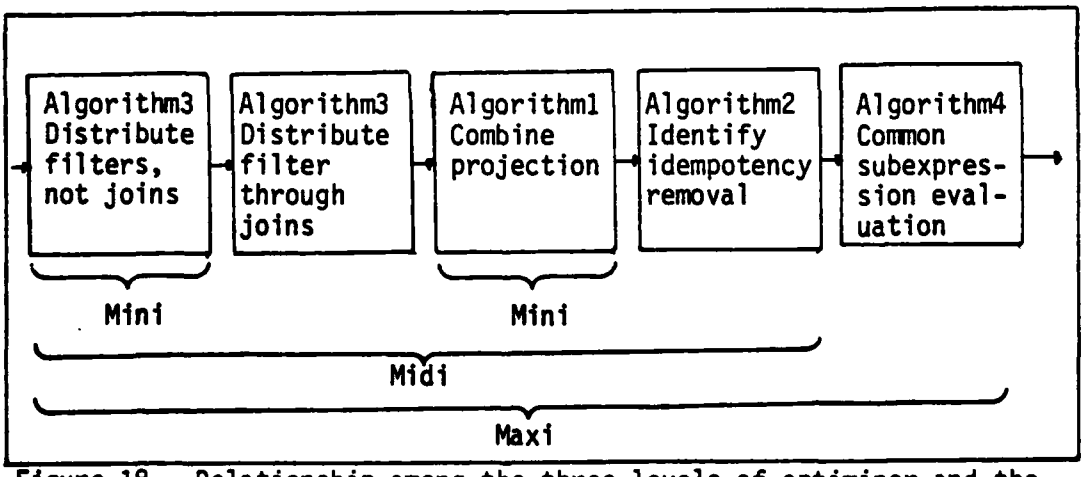


Figure 18. Relationship among the three levels of optimizer and the transformations.

VI. CONCLUSION

OVERVIEW

The title of this thesis implies that two goals were sought. One was to design a database system for pedagogical applications, the other to design a relational database system using the most efficient implementation methods available. The first goal was met to a great extent in the design of the data manipulation and definition language. The decision to use a relational algebra based design was influenced by this goal as well as the provisions to allow each user to be a kind of "mini-" database administrator, defining and controlling distinct sets of relations.

The second goal was achieved in that a relational system was designed, however, the efficiency of said system is yet to be conclusively proven. Even so, a series of transformations that can be applied to a relational query to produce an equivalent query that can be executed faster has been presented. These transformations are in the strictest sense of the word not optimizing but rather ameliorating because they can not be guaranteed to improve the time taken to compute the result. However, as argued, the chosen transformations are reasonably likely to improve performance.

An important consideration is the interaction between these goals. Since the system is earmarked for use in a pedagogical environment, the level of use will many times be very low; i.e., small queries on small relations. Thus, in these cases the optimization procedures should be "turned off", either in part or completely, in order to eliminate their overhead without affecting the user's viewpoint of overall system performance.

FUTURE RECOMMENDATIONS

Due to the limited development life of this thesis and the expanse of computing milue which this thesis addresses, there are many opportunities for future projects and/or follow on thesis investigations. These recommendations are:

(1) Efficiently implement the low-level access modules using the basic procedures outlined in Appendix C and the generalized access path structure discussed in Chapter IV, keeping in mind the limits on space and time imposed by the use of a micro-computer system.

(2) Once the low-level modules are implemented, a major project would involve validating the optimization procedures and fine-tuning them to exact optimal behavior from the overall system.

(3) Alternatives to the operator tree structure may be examined.

(4) Cost estimation methods for determining when to apply the optimization procedures should be examined.

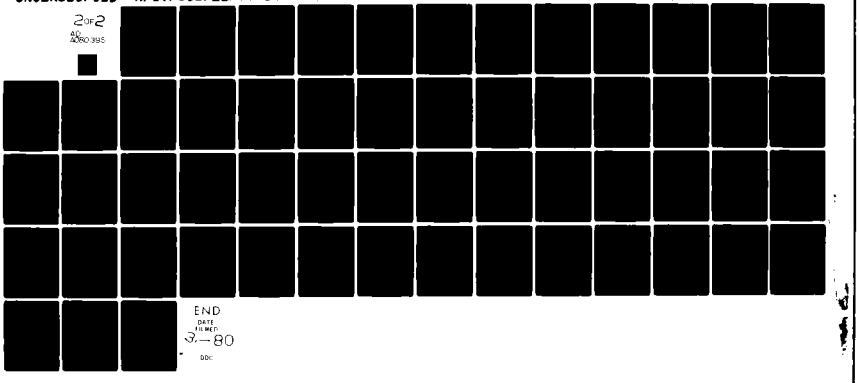
(5) Once the system is completed, introduction of a sub-schema definition (Ref 2) level between the data entry and conceptual levels may be a desired and useful addition.

(6) Consideration should be given to developing a dual data entry system for educated users. For example, cutting down of the length and/or number of prompts, automatic correction of simple errors, etc.

FINAL COMMENT

Database systems defined using the relational view have great potential for revolutionizing the information industry. The relational view promises a simple, flexible approach to a person or business's information retrieval problem; if only the problem of efficiency can

AD-A080 395 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 5/2
THE DESIGN AND IMPLEMENTATION OF A PEDAGOGICAL RELATIONAL DATAB--ETC(U)
DEC 79 M A ROTH
UNCLASSIFIED AFIT/GCS/EE/79-14 NL



be solved. It may be that technology may solve the problem by making computers bigger, faster, and cheaper so that what is inefficient now will be tolerable later. However, even larger improvements can be made by improving software which is currently a generation of growth behind the hardware on which it runs.

BIBLIOGRAPHY

1. Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, 13 (6): 377-387 (June 1970).

2. Date, C.J., An Introduction to Database Systems (Second Edition). Reading: Addison-Wesley, 1977.

3. Banerjee, Jayanta, et al. "DBC -- A Database Computer for Very Large Databases," IEEE Transactions on Computers, 28 (6): 414-429 (June 1979).

4. CP/M User's Guide, Digital Research. (Available from AFIT/ENE).

5. UCSD (Mini-Micro Computer) PASCAL, Version II.0, Institute for Information Systems, University of California, San Diego (March 1979). (Available from AFIT/ENE).

6. Whitney, Kevin M., "Relational Data Management Implementation Techniques," Proceedings 1974 ACM SIGMOD Workshop on Data Description, Access, and Control, New York (1974).

7. Palermo, F.P., "A Data Base Search Problem", Proceedings of the COINS-72) Symposium (December 1972).

8. Nijssen, G.M., "Present and Future Possibilities of Database Technology," Proceedings of the IFIP Congress, Stockholm (1974).

9. Codd, E.F., "Relational Completeness of Database Sublanguages," Courant Computer Science Symposium 6 Data Base Systems, 321-328, New York (1974).

10. Popa, J.H., "Relational Data Management", Department of Defense, 1976 (AD A029892).

11. Chamberlin, D.D. and R.F. Boyce "SEQUEL: A Structured English Query Language," Proceedings 1974 ASM SIGMOD Workshop on Data Description, Access, and Control, New York (1974).

12. Smith, J.M., and P.M.-T. Chang "Optimizing the Performance of a Relational Algebra Database Interface," Communications of the ACM, 18 (10): 568-579 (October 1975).

13. Eswaran, Kapali P. and Donald D. Chamberlin "Functional Specifications of a Subsystem for Data Base Integrity,"Proceedings International Conference on Very Large Data Bases (September 1975).

14. Hall, P.A.V., "Optimization of Single Expressions in a Relational Data Base System," IBM Journal of Research and Development, 20 (3): 244-257 (1976)

15. Haerder, Theo "A Generalized Access Path Structure," ACM Transactions on Database Systems, 3 (3): 285-298 (September 1978).

16. Bayer, R. and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," Acta Infromatica 1 (3): 173-189 (1972).

17. Wedekind, H., "On the Selection of Access Paths in a Data Base System," In Data Base Management, J.W. Klimbie and K.L. Koffeman, Eds. Amsterdam: North-Holland Publishing Company, 1974.

18. Blasgen, M.W., et al., "An Encoding Method for Multi-field Sorting and Indexing," IBM Research Report RJ1753, IBM Research Laboratory, San Jose, California (March 1976).

19. Astrahan, M.M. and D.D. Chamberlin, "Implementation of a Structured English Query Language," Communications of the ACM, 18 (10): 580-588 (October 1975).

20. Chang, P.V., "A design for a relational database system," University of Utah Technical Report.

APPENDIX A

A COMMUNICATION NETWORK BETWEEN THE ALTAIR 8080 AND THE INTEL 8080 MICROPROCESSOR SYSTEMS

The steps used in bringing up CP/M and consequently UCSD Pascal on the ALTAIR 8080 system using the existing facilities of CP/M on the INTEL 8080 system are outlined below.

1) The first step was to design a communication network between the Intel and Altair. A three way RS-232 interconnection box was designed. The schematic for this box is shown in Figure A-1. The cable connections were made as follows:

Cable 1 -- Intel serial CH2

Cable 2 -- Altair serial Port 1

Cable 3 -- CRT Modem Port

The CRT to which Cable 3 was connected was used to talk to the Intel. Another CRT was needed to talk to the Altair. Thus a separate RS-232 cable was connected to the second CRT's Modem Port and to the Altair, serial Port Ø.

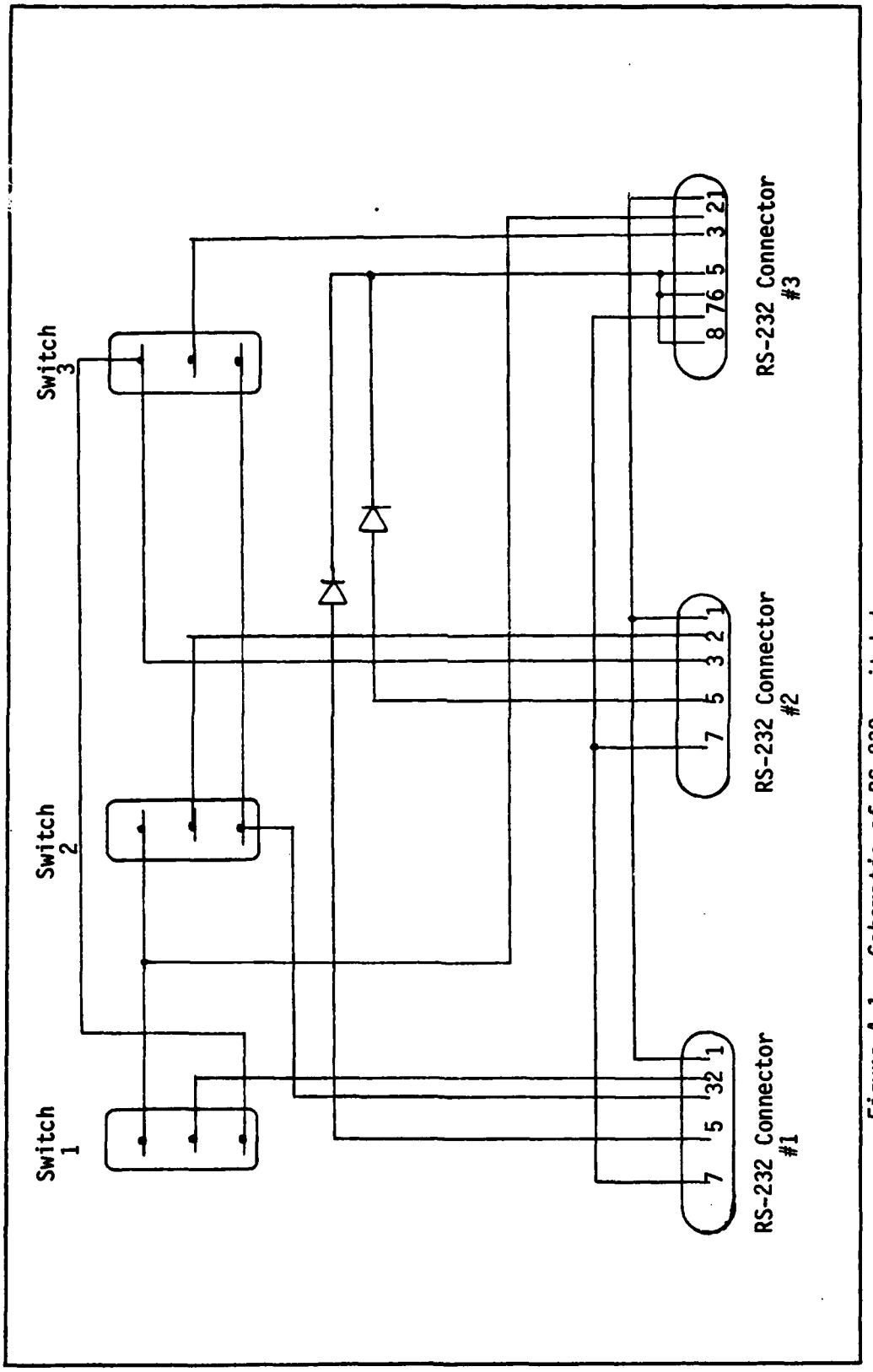
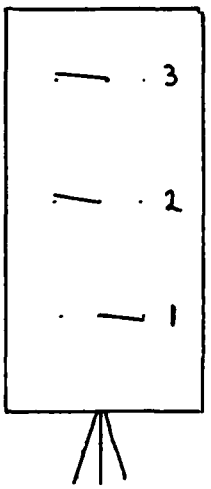


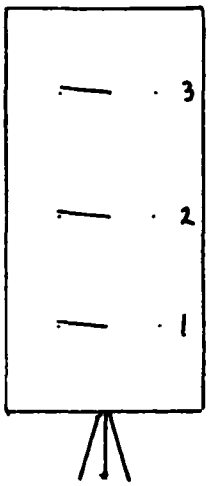
Figure A-1. Schematic of RS-232 switch box.

The following two configurations for the switches were used in this project:

Configuration 1



Configuration 2



Configuration 1 allows the systems to act independently, but with input to the Altair allowed via Port 1 from the Intel machine. Configuration 2 forces the Intel system to accept inputs from the Altair rather than its CRT. Input is still allowed to the Altair via Port 1.

2) The next step was to create two programs, one to run on the Altair, the other on the Intel, with the purpose of transferring a CP/M core image to the Altair and then forcing the Altair to execute that image. This CP/M image will force the Altair to access the disks of the Intel system over serial Port 1. Thus a program to make the Intel think it was a disk was designed. Under the purview of this program the Intel's sole purpose was to do disk I/O for the Altair. When the Altair wished

to read a sector it would send an 'R' over the serial line. This would tell the Intel to prepare for a read, receiving the disk, track, and sector numbers the Altair was now sending it. The Intel would then access the disk and send the information back to the Altair over the serial line. Similar actions occurred for a disk write. In addition, since the Altair is now accessing the Intel disk, when a reboot is necessary, the disk which it reboots from must contain the same CP/M image passed to the Altair. This can be done using the SYSGEN program, by first building the image in core and then via SYSGEN putting it on the disk.

The programs to do this were CMP16K -- program on Intel to transfer CP/M image, BOOTCPM -- program on Altair to get core image and transfer control to it, and DISKSIM -- program on intel to simulate a disk over the serial lines. The source code for BOOTCPM and DISKSIM are attached; however, the source code for CPM16K and the disk which was created to reboot CP/M under control of the Altair was destroyed in one of several disk crashes not related to this effort. A description of the content of CPM16K appears below.

The CPM16K program contains a CP/M core image with a BIOS which will force the Altair to perform disk accesses over serial Port 1. The actual executable program does the following:

1. Send a start character to the Altair; this is the character which BOOTCPM waits for.

2. Send the address where the Altair is to start storing the CP/M image, the length of the image and an address for the Altair to go to to start executing the CP/M.

3. Send the CP/M image.

4. Quit.

The following steps were used in executing the above procedures:

On Intel CRT:

0) Set switches on box to Configuration 1.

1) Bring CP/M up on the Intel system.

2) Change baud rate of Intel serial CH 1 to 9600 baud.

3) Prepare to execute CPM16K; i.e., type CPM16K, but no carriage return.

On Altair CRT:

4) Bring the Altair operating system up.

5) Mount disk 0; i.e., type MNT Ø <cr>.

6) Execute linker; i.e., type LINK <cr>.

7) Link in BOOTCPM; i.e., type L BOOTCPM Ø <cr>.

8) Execute BOOTCPM; i.e., type X <cr>.

On Intel CRT:

9) Execute CPM16K; i.e., type <cr>.

10) Execute DISKSIM; i.e., type DISKSIM cr and immediately change switch settings to Configuration 2. It not done soon enough, the entire process must be repeated.

11) *The CP/M prompt should appear on the Altair CRT.*

3) The next goal was to create a CP/M disk for the Altair system. Using the facilities of CP/M brought up with the above procedures, programs to perform disk I/O for the Altair disks were created. These programs were patched into the BIOS of the CP/M image so that then access could be made to either the Intel disks over the serial lines or the Altair disks using its own disk access ports. Using this new version of CP/M, a program to read in tracks 1 and 2 was placed on track Ø, sector 1 of the future CP/M disk. Then using the CP/M utility, SYSGEN, a copy of the CP/M image was placed on the disk. SYSGEN places this copy on tracks 1 and 2. At that point a complete CP/M disk existed for the Altair, and only a method of booting that disk was needed. Thus a program was created, to reside in PROM, which would first, copy the I/O for doing a

disk access from the PROM down to location 3ØØØH (hex address), so that the boot could use it, and then read in track Ø, sector 1 from the disk, and finally transfer control to that program.

The programs used in this step are ZERO -- the program residing on track Ø, sector 1, and ALT-ROM the program put in PROM. The source code for these programs are attached.

4) Now that CP/M existed as a stand alone system on the Altair, USCD Pascal could be easily implemented. Since the disk simulation program was still available, and the BIOS for the Altair CP/M still included the functions necessary to use it, a Pascal system disk was copied from the soft-sectored disk that the Intel uses to the hard-sectored disk the Altair uses. With appropriate modifications for the type of terminal being used, Pascal was then functional.

The procedures to be used now to bring CP/M up on the Altair with access to both its own disks and the Intel disks are as follows:

1) Set switches on box to Configuration 1.

2) Bring up CP/M in Intel.

3) Bring up CP/M on Altair by executing the program in PROM located at F8ØØH.

4) Execute DISKSIM on Intel.

5) Switch box to Configuration 2.

6) Disk access is as follows:

A: Altair drive 0

B: Altair drive 1

C: Intel drive Ø

D: Intel drive 1.

7) The Intel CRT can be disconnected if desired and the upper switch on the box set to the neutral or middle position.

```
; BOOTCPM PROGRAM WHICH RUNS ON ALTAIR
STADR   EQU     200H
STAK    EQU     300H
        ORG     0F800H
LSET    EQU     11H
SERP    EQU     13H
SERS    EQU     12H
RBR     EQU     1
TXRDY   EQU     2
BOOT    MVI     A,LSET
        OUT     SERS
BLOOP   LXI     SP,STAK
        CALL    GETCH
        CPI     ':'
        JNZ     BLOOP
        CALL    GETAD
        SHLD    STADR
        CALL    GETAD
        XCHG
        CALL    GETAD
GLOOP   CALL    GETCH
        MOV     M,A
        INX     H
        DCX     D
        MOV     A,D
        ORA     E
        JNZ     GLOOP
        LHLD    STADR
        PCHL
GETAD   CALL    GETCH
        MOV     L,A
        CALL    GETCH
        MOV     H,A
        RET
GETCH   IN      SERS
        ANI     RBR
        JZ      GETCH
        IN      SERP
        RET
        END
```

```
;       PROGRAM DISKSIM, RUNS ON THE INTEL
;       THIS PROGAM SIMULATES THE EXISTANCE OF A DISK.
;       THROUGH A PROTOCOL OVER THE LINES IT WILL
;       ACCESS THE DISK FOR THE REQUESTING COMPUTER.
;
        ORG     100H
STAK        EQU     1000H
DBUF        EQU     2000H
    LXI     SP,STAK
    LXI     B,DBUF
    CALL    SETDMA
    MVI     A,':'           ; SEND A BAD BLOCK FIRST
    MVI     B,130           ; 128+ START+CS
SLOOP
    CALL    SENDCH
    DCR     B
    JNZ     SLOOP
;
FDLOOP
    CALL    GETCH
    CPI     'R'     ; CHECK FOR AN 'R'
    JNZ     WRITETEST
    CALL    GETDTS
    MVI     A,'R'
    CALL    SETDTS
    CALL    READ
    MVI     E,0
    LXI     H,DBUF
    MVI     B,128
    MVI     A,':'
    CALL    SENDCH
RLOOP
    MOV     A,M
    CALL    SENDCH
    ADD     E
    MOV     E,A     ; DO CHECKSUM
    INX     H
    DCR     B
    JNZ     RLOOP
    MOV     A,E     ; SEND CSUM
    CALL    SENDCH
    JMP     FDLOOP
WRITETEST
    CPI     'W'
    JNZ     FDLOOP  ; IF NOT TRY AGAIN
    CALL    GETDTS  ; GET VALUES FOR DISK TRK SECTOR
    MVI     E,0     ; CHECKSUM
    MVI     B,128
    LXI     H,DBUF
```

```
WLOOP
    CALL    GETCH
    MOV     M,A
    INX     H
    ADD     E
    MOV     E,A
    DCR     B
    JNZ     WLOOP
    CALL    GETCH
    SUB     E       ; SUBTRACT CHECKSUM
    JNZ     BADTRANS
    MVI     A,'W'           ; SIGNAL FOR WRITE
    CALL    SETDTS
    CALL    WRITE
    MVI     A,'G'
    CALL    SENDCH
    JMP     FDLOOP
BADTRANS
    MVI     A,'B'
    CALL    SENDCH
    JMP     FDLOOP
RBR EQU     2
TXRDY       EQU     1
CONST       EQU     0F7H
CONIN       EQU     0F6H
CONOUT      EQU     0F6H
SENDCH
    PUSH    H
    PUSH    B
    PUSH    D
    PUSH    PSW
PLOOP       IN      CONST
    ANI     TXRDY
    JZ      PLOOP
    POP     PSW
    OUT     CONOUT
    POP     D
    POP     B
    POP     H
    RET
GETCH
    IN      CONST
    ANI     RBR
    JZ      GETCH
    IN      CONIN
    RET
```

```
;
;*****************************************
;
;           GET DTS DISK, TRACK, SECTOR
;
GETDTS
    CALL    GETCH
    STA     DISK
    CALL    GETCH
    STA     TRACK
    CALL    GETCH
    STA     SECTOR
    RET
;
DISK        DB      0
TRACK       DB      0
SECTOR      DB      1
;
;
SETDTS
    STA     PRTYPE          ; STORE A IN PRINT STRING
    LDA     DISK            ;
    MOV     C,A             ; PUT DISK NUM IN C FOR SELDSK
    CALL    CONVHEX
    SHLD    PRDISK
    CALL    SELDSK
    LDA     TRACK
    MOV     C,A             ; PUT TRACK IN C FOR SETTRK
    CALL    CONVHEX         ; PUT HEX VAL OF A IN HL
    SHLD    PRTRACK
    CALL SETTRK
    LDA     SECTOR
    MOV     C,A
    CALL    CONVHEX
    SHLD    PRSECTOR
    CALL    SETSEC
    JMP     POUT

STRING      DB      0AH,0DH
PRTYPE      DB      'AD='
PRDISK      DB      '00'
PRTRACK     DB      '00'
PRSECTOR DB '00'
    DB      0,0,0,0

; PRINT OUT ACCESS STRING
POUT
    LXI     H,STRING
```

```
PROUTLOOP
    MOV     A,M
    ANA     A               ; CHECK FOR ZERO
    RZ
    MOV     C,A
    PUSH    H
    CALL    LST
    POP     H
    INX     H
    JMP     PROUTLOOP

CONVHEX
    PUSH    PSW
    PUSH    D
    PUSH    B
    MOV     B,A
    ANI     0FH
    CALL    FIXNUM
    MOV     H,A
    MOV     A,B
    RAR
    RAR
    RAR
    RAR
    ANI     0FH
    CALL    FIXNUM
    MOV     L,A
    POP     B
    POP     D
    POP     PSW
    RET
FIXNUM
    ADI     '0'
    CPI     '9'+1
    RM
    ADI     'A'-'0'-10
    RET
```

```
;*****************************************************
;
;           CPM INTERFACE ROUTINES
;
;*****************************************************

SELDSK:     LHLD    0001H           ; GET BIOS ADDR
    MVI     L,1BH
    PCHL
SETSEC:     LHLD    0001H
    MVI     L,21H
    PCHL
SETTRK:     LHLD    0001H
    MVI     L,1EH
    PCHL
SETDMA:     LHLD    0001H
    MVI     L,24H
    PCHL
READ:       LHLD    0001H
    MVI     L,27H
    PCHL
WRITE:      LHLD    0001H
    MVI     L,2AH
    PCHL
HOME:       LHLD    0001H
    MVI     L,18H
    PCHL
LST
    RET
    LHLD    1
    MVI     L,0FH
    PCHL

;*********************************************
    END
```

```
; ZERO PROGRAM WHICH RESIDES ON TRACK 0, SECTOR 1
; OF ALTAIR CP/M DISK, USED TO BOOT
;
        ORG     0
MEMAD   EQU     0A500H      ; ADDR TO READ TRACK 1 AND 2 INTO
DONE    EQU     0BA00H      ; PLACE TO GOTO IN CP/M WHEN DONE
        LXI     H,MEMAD     ; READ IN ADDRESS
        LXI     D,1         ; BLOCK NUMBER
        MVI     C,25        ; SECTOR COUNT FOR TRACK 1
LOOP    MVI     B,0         ; DISK NUMBER
        PUSH    D
        PUSH    H
HERE    CALL    3000H       ; READ A SECTOR USING CP/M BIOS
        ORA     A           ; SEE IF ERROR ON READ
        JNZ     HERE        ; IF SO TRY READING AGAIN
        POP     H
        LXI     D,128       ; BUMP MEM ADDR ANOTHER SECTOR
        DAD     D
        POP     D
        INX     D           ; NEXT SECTOR
        DCR     C           ; DONE WITH TRACK 1?
        JNZ     LOOP
        MOV     A,E         ; SEE IF DID TRACK 2
        CPI     28
        JP      DONE        ; IF SO START CPM-ING
        LXI     D,32        ; ELSE START NEXT TRACK
        MVI     C,26
        JMP     LOOP
        END
```

```
; ALT-ROM PROGRAM WHICH RESIDES IN PROM ON THE
; ALTAIR AT LOCATION F800, BOOTS CP/M
ROM     EQU     0F800H
LOC     EQU     3000H
DELTA   EQU     -19H
BEG     ORG     LOC+DELTA
        LXI     SP,3FFFH
        LXI     H,ROM-DELTA
        LXI     D,START
        LXI     B,NB
ONW     MOV     A,M
        STAX    D
        INX     H
        INX     D
        DCX     B
        MOV     A,B
        ORA     C
        JNZ     ROM+ONW-BEG
        JMP     START2
START                   ;ENTRY TO READ BLOCK IN DE,ADDR IN HL,DISK IN B
        JMP     READ    ;ALLOWS EASY INTERFACE TO OTHER BOOT
START2                  ;ENTRY TO READ TRACK 0 SECTOR 1
        MVI     B,0
        LXI     D,0     ;TRACK 0 SECTOR 0
        LXI     H,0     ;READ INTO LOC 0
        MVI     A,129
        CALL    READ
        JMP     0
READ
        SHLD    DMA     ;SAVE DMA ADDR
        MVI     A,129   ;NUMBER OF BYTES TO READ
        LXI     H,MYDMA ;LOC OF READ BUFFER
        CALL    READA   ;DO DISK IO
        ORA     A       ; SEE IF ERROR
        RNZ             ; IF SO RETURN
        MVI     B,0
        LHLD    DMA
        LXI     D,MYDMA
        MVI     C,128
EXCH    LDAX    D
        MOV     M,A
        XRA     B
        RRC
        MOV     B,A
        INX     D
        INX     H
        DCR     C
        JNZ     EXCH
        LDAX    D
```

```
        XRA     B       ;COMPARE WITH CRC
        JNZ     BADCRC
        XRA     A       ;ZERO A
        RET
BADCRC  MVI     A,0     ; INDICATE BAD CRC
        RET
; DISK HANDLING ROUTINE FOR THE ALTAIR DISK DRIVE
;
; ENTER AT READ OR WRITE WITH THE FOLLOWING VALUES
; IN REGISTERS A,B,DE,AND HL.
;
; REG A CONTAINS NUMBER OF BYTES TO BE INPUT OR OUTPUT
; REG B CONTAINS THE DISK DRIVE NUMBER
; REG DE CONTAINS THE BLOCK NUMBER RANGING FROM 0
;       TO 2463 DECIMAL
; REG HL CONTAINS THE BUFFER ADDRESS WHERE DATA IS TO
;       BE READ FROM OR WRITTEN TO
;
; ALL REGISTERS ARE RETRUNED INTACT
; INTERRUPTS ARE NOT ALLOWED DURING DISK I/O AND THE
; INTERRUPTS ARE ENABLED ON RETURN
;
;
; SUBROUTINE TO ENABLE DISK DRIVE
;
ENAB    LXI     H,TRKNM         ; POINT TO START OF TRACK TABLE
        MOV     A,B             ; PUT DRIVE NUMBER IN REG A
ENAB1   DCR     A               ; INCREMENT HL SO THAT IT WILL
        JM      ENAB2           ;  POINT TO THE CORRECT STATUS BYTE
        INX     H               ;
        JMP     ENAB1           ;
ENAB2   MOV     A,M             ; GET STATUS BYTE FOR SPECIFIED DRIVE
        ANI     80H             ; TEST IF IT HAS BEEN INITIALIZED
        JNZ     TRK0            ; IF NOT THEN INTIALIZE IT
        LDA     DRVNM           ; GET CURRENT DRIVE NUMBER
        CMP     B               ; IS IT THE SAME AS REQUESTED DRIVE?
        JNZ     ENAB3           ; IF NOT THEN ENABLE NEW DRIVE
        IN      DSTAT1          ; CHECK IF CURRENT DRIVE IS ENABLED
                                ; IN CASE DRIVE DOOR HAS BEEN OPENED
        ANI     ENBIT           ;
        RZ                      ; RETURN IF ENABLED
ENAB3   MOV     A,B             ; GET DRIVE NUMBER
        STA     DRVNM           ; AND SAVE IT
        OUT     DSTAT1          ; ENABLE DRIVE
        IN      DSTAT1          ; IS DRIVE ENABLED NOW?
        ANI     ENBIT           ;
        JNZ     ENAB3           ; IF NOT KEEP TRYING
        RET
```

```
;
; SUBROUTINE TO FIND TRACK 0
;
TRK0    CALL    ENAB3           ; ENABLE DISK DRIVE
TRK01   CALL    MOVE            ; BE SURE HEAD MOVE IS ALLOWED
                                ; TO GAURANTEE HEAD IS SETTLED
        IN      DSTAT1          ; GET DISK STATUS
        ANI     TOBIT           ; CHECK FOR TRACK 0
        JZ      TRK02           ; GO SET STATUS IF ON TRACK 0
        CALL    OUT1            ; IF NOT 0, STEP OUT
        JMP     TRK01           ; CHECK AGAIN FOR TRACK 0
TRK02   MOV     M,A             ; SET DISK STATUS BYTE TO INDICATE
                        ; ZERO AND THAT DRIVE HAS BEEN INITIALIZED
;
; SUBROUTINE TO STEP HEAD OUT
;
OUT1    CALL    MOVE            ; BE SURE HEAD MOVE IS ALLOWED
        MVI     A,OUTBT         ; GET STEP OUT COMMAND
        OUT     DCONT           ; STEP HEAD OUT
        RET
;
; SUBROUTINE TO STEP HEAD IN
;
IN1     CALL    MOVE            ; BE SURE HEAD MOVE IS ALLOWED
        MVI     A,INBIT         ; GET STEP IN COMMAND
        OUT     DCONT           ; STEP HEAD IN
        RET
;
; SUBROUTINE TO FIND TRACK N AND SET THE
; HEAD CURRENT SWITCH STATUS
;
TRKN    MOV     A,M             ; GET CURRENT TRACK NUMBER
        CMP     B               ; CHECK FOR DESIRED TRACK
        JZ      STHCS           ; IF EQUAL GO SET HEAD CURRENT SWITCH
        JC      MVIN            ; STEP HEAD IN IF B IS GREATER THAN A
MVOUT   DCR     A               ; DECREMENT TRACK NO
        MOV     M,A             ; AND SAVE IT
        CALL    OUT1            ; AND STEP HEAD OUT
        JMP     TRKN            ; CHECK AGAIN FOR CORRECT TRACK
MVIN    INR     A               ; INCREMENT TRACK NO
        MOV     M,A             ; AND SAVE IT
        CALL    IN1             ; AND STEP HEAD IN
        JMP     TRKN            ; CHECK AGAIN FOR CORRECT TRACK
STHCS   CPI     43              ; IS TRACK GT OR EG TO 43?
        MVI     A,HCSON         ; GET HEAD CURRENT ON COMMAND
        JNC     STHC            ; IF GT OR EQ TO 43 SET HEAD CURRENT SWITCH
        MVI     A,HCOFF         ; GET HEAD CURRENT OFF COMMAND IF LT 43
STHC    STA     HCS             ; SAVE HEAD CURRENT SWITCH STATUS
        RET
```

```
;
; SUBROUTINE TO DEDUCE TRACK AND SECTOR NUMBERS
; FROM LOGICAL RECORD NUMBER SUPPLIED BY CALLING ROUTINE
;
RECRD   MOV     A,E             ; GET LOW BYTE OF RECORD NUM
        ANI     1FH             ; ISOLATE LOW FIVE BITS (SECTOR ADDR)
        MOV     C,A             ; SAVE SECTOR ADDRESS IN C
        MOV     A,E             ; GET LOW BYTE AGAIN
        CALL    ROT             ; ROTATE TO GET LOW THREE BITS OF TRK NUM
        ANI     07H             ; ISOLATE THOSE THREE DATA BITS
        MOV     B,A             ; SAVE THEM IN B
        MOV     A,D             ; GET HIGH BYTE OF RECORD NUM
        CALL    ROT             ; JUSTIFY IT AS ABOVE
        ANI     0F8H            ; ZERO LOW THREE BITS
        ORA     B               ; OR IN THE LOW THREE BITS
        CPI     78              ; CHECK FOR VALID TRACK
        JNC     ERR1            ; JUMP IF ERROR
        MOV     B,A             ; SAVE TRACK NUMBER IN B
        STA     TEMP            ;
        RET
;
; SUBROUTINE TO ROTATE THREE TIMES
;
ROT     RLC
        RLC
        RLC
        RET
;
; SUBROUTINE TO WAIT TILL HEAD MOVE IS ALLOWED
;
MOVE    IN      DSTAT1          ; GET DISK STATUS
        ANI     MVBIT           ; CHEC HEAD MOE BIT
        JNZ     MOVE            ; WAIT TILL MOVE IS ALLOWED
        RET
;
; SUBROUTINE TO SAVE REGISTERS AND INVOKE SUPPORT ROUTINES
; TO PREPARE DISK FOR READ OR WRITE
;
INIT    SHLD    BUFAD           ; SAVE BUFFER ADDRESS
        XTHL                    ; PUSH HL AND GET RETURN ADDRESS
        PUSH    D               ; SAVE OTHER REGISTERS
        PUSH    B
        PUSH    PSW
        PUSH    H               ; PUT RETURN ADDRESS BACK ON STACK
        STA     BYTES           ; SAVE BYTE COUNT
        CALL    ENAB            ; ENABLE DISK DRIVE
        CALL    RECRD           ; GET TRACK AND SECTOR NUMBERS
        CALL    TRKN            ; POSITION HEAD
        LHLD    BUFAD           ; GET BUFFER ADDRESS BACK IN HL
```

```
        IN      DSTAT1          ; GET DISK STATUS
        ANI     HDBIT           ; CHECK IF HEAD IS LOADED
        JZ      OKAY            ; SKIP LOAD IF ALREADY LOADED
        MVI     A,HDBIT         ;
        OUT     DCONT           ; LOAD HEAD
OKAY    LDA     BYTES           ; GET BYTE COUNT
        MOV     D,A             ; AND KEEP IN D
        DI                      ; DON'T ALLOW INTERRUPTS DURING DISK I/O
;
; SUBROUTINE TO FIND START OF DESIRED SECTOR
;
        IN      POS             ; READ SECTOR POSITION STATUS
        MOV     B,A             ; SAVE IN B
        ANI     SECBT           ; TEST FOR START OF SECTOR
        JNZ     SEC             ; IF NOT START TRY AGAIN
        MOV     A,B             ; GET SECTOR NUMBER IN ACCUM
        RAR                     ; JUSTIFY IT
        ANI     1FH             ; ISOLATE ADDRESS BITS
        CMP     C               ; CHECK FOR DESIRED SECTOR
        RZ                      ; RETURN TO READ OR WRITE IF CORRECT SECTOR
        JMP     SEC             ; IF NOT , TRY AGAIN
;
; ENTRY POINT TO READ A SECTOR
;
READA   CALL    INIT            ; GET READY TO READ
RSYN    IN      DSTAT1          ; GET DISK STATUS
        RAL                     ; SHIFT DATA AVAILABLE BIT TO CARRY
        JC      RSYN            ; LOOP TILL DATA IS READY
        IN      DATA            ; READ SYNC BYTE
FB      IN      DSTAT1          ; GET DISK STATUS
        RAL                     ; SHIFT TO CARRY
        JC      FB              ; KEEP LOOKING FOR FIRST BYTE
        IN      DATA            ; READ FIRST BYTE
RDAT1   MOV     E,A             ; PUT DATA IN E SO READ ROUTINE WILL WORK
RDAT    IN      DSTAT1          ; GET DISK STATUS
        RAL                     ; SHIFT TO CARRY
        JC      RDAT            ; KEEP LOOKING FOR DATA READY
        IN      DATA            ; READ DATA
        MOV     M,E             ; STORE FIRST BYTE
        INX     H               ; POINT T NEXT BYTE IN BUFFER
        DCR     D               ; DECREMENT BYTE COUNT
        JZ      EXIT            ; EXIT IF DONE
        MOV     M,A             ; IF NOT DONE. STORE THIS BYTE IN BUFFER
        INX     H               ; POINT TO NEXT BYTE IN BUFFER
        DCR     D               ; DECREMENT BYTE COUNT
        IN      DATA            ; TIME TO READ NEXT BYTE FROM DISK
        JNZ     RDAT1           ; READ MORE IF NOT DONE
```

```
; ROUTINE TO LEAVE DISK HANDLER
;
EXIT    POP     PSW             ; RESTORE REGISTERS
        POP     B
        POP      D
        POP     H
        XRA     A
        RET                     ; GO BACK TO CALLING ROUTINE
;
; APPROPRIATE ERROR MESSAGE IF YOU HAVE THAT CAPABILITY
;
ERR1    MVI     A,1
        POP     D
        POP     B
        POP     D
        POP     H
        RET
;
;
; STORAGE REQUIRED BY DISK HANDLER
; MAY BE ORG'D TO ANYWHERE YOU HAVE MEMORY IF
; YOU WANT THE DISK PROGRAM IN PROM
;
DRVNM   DB      0               ; CONTAINS CURRENT DRIVE NUMBER
TRKNM   DB      80H             ; TABLE OF STATUS BYTES FOR DRIVES 0-3
        DB      80H             ; STATUS BYTES ARE INITIALY SET OT 80 HEX
        DB      80H             ; SO ENABLE ROUTINE WILL INIT DISK
        DB      80H             ; DRIVES THE FIRST TIME THEY ARE USED
; AFTER INITIALIZATION THE STATUS BYTE HOLDS THE
; CURRENT TRACK POSITION FOR ITS DRIVE
HCS     DB      80H             ; STATUS FOR HEAD CURRENT SWITCH
BYTES   DB      0               ; TEMPORARY SAVE SPACE FOR NUMBER
                                ;  OF BYTES TO BE INPUT OR OUTPUT
DMA     DW      00              ; SPACE FOR READ/WRITE BUFFER ADDR
BUFAD   DW      0               ; TEMP SAVE SPACE FOR BUFFER ADDR
TEMP    DB      0FFH            ;
MYDMA   EQU     $+5             ; BUFFER FOR READ AND WRITE
```

```
;
;
; EQUATES
;
NB      EQU     200H            ; NUMBER OF BYTES IN PROGRAM
DSTAT1  EQU     8               ; DISK STATUS PORT
DCONT   EQU     9               ; DISK CONTROL PORT
POS     EQU     9               ; SECTOR POSITION PORT
DATA    EQU     10              ; DATA PORT
HDBIT   EQU     4               ; HEAD CONTROL AND TEST BIT
ENBIT   EQU     8               ; DISK ENABLED TEST BIT
TOBIT   EQU     40H             ; TRACK 0 TEST BIT
INBIT   EQU     1               ; STEP HEAD IN COMMAND
OUTBT   EQU     2               ; STEP HEAD OUT COMMAND
MVBIT   EQU     2               ; HEAD MOVE TEST BIT
HCSON   EQU     0C0H            ; HEAD CURRENT SWITCH AND WRITE ENABLE
HCOFF   EQU     80H             ; WRITE ENABLE AND HEAD CURRENT SWITCH OFF
SECBT   EQU     1               ; START OF SECTOR TEST BIT
SYNC    EQU     0FFH            ; SYNC BYTE
END
```

APPENDIX B

USER'S GUIDE: THE ROTH RELATIONAL DATABASE SYSTEM

INTRODUCTION AND OVERVIEW * SECTION 1

The database system (hereafter referred to as "the system") described in this document is a system intended to run on stand alone micro- and mini-computers under the control of the UCSD Pascal operating system, Version II.0. All system software is written in Pascal, resulting in relatively straightforward software maintenance and enhancement.

The system is designed to be used primarily with a CRT terminal as the CONSOLE device; however, the system is flexible enough to be reconfigured for slower hard-copy terminals. The system does require some kind of fast mass storage such as a floppy disk system or better. The initial development of the system was done on an Intel 8080 microprocessor with dual-drive floppy disks and an ADM-3A CRT terminal.

1.1 The Database System: An Overview

The structure of the system is best conceptualized in terms of the "tree-like" structure diagram in Figure B-1.

The diagram in Figure B-1 depicts the outermost level of the system. In terms of a "tree" or structure diagram, the "root" corresponds to the outermost level, while the "leaves" (i.e., the triangles with no branches to lower levels) correspond to the lower levels of the system. While a user is in a particular level, the system displays a list of available commands called the "prompt-line." If the system is running on a CRT terminal, then the prompt-line will usually appear at the top of the screen. Commands are usually invoked by typing a single character from

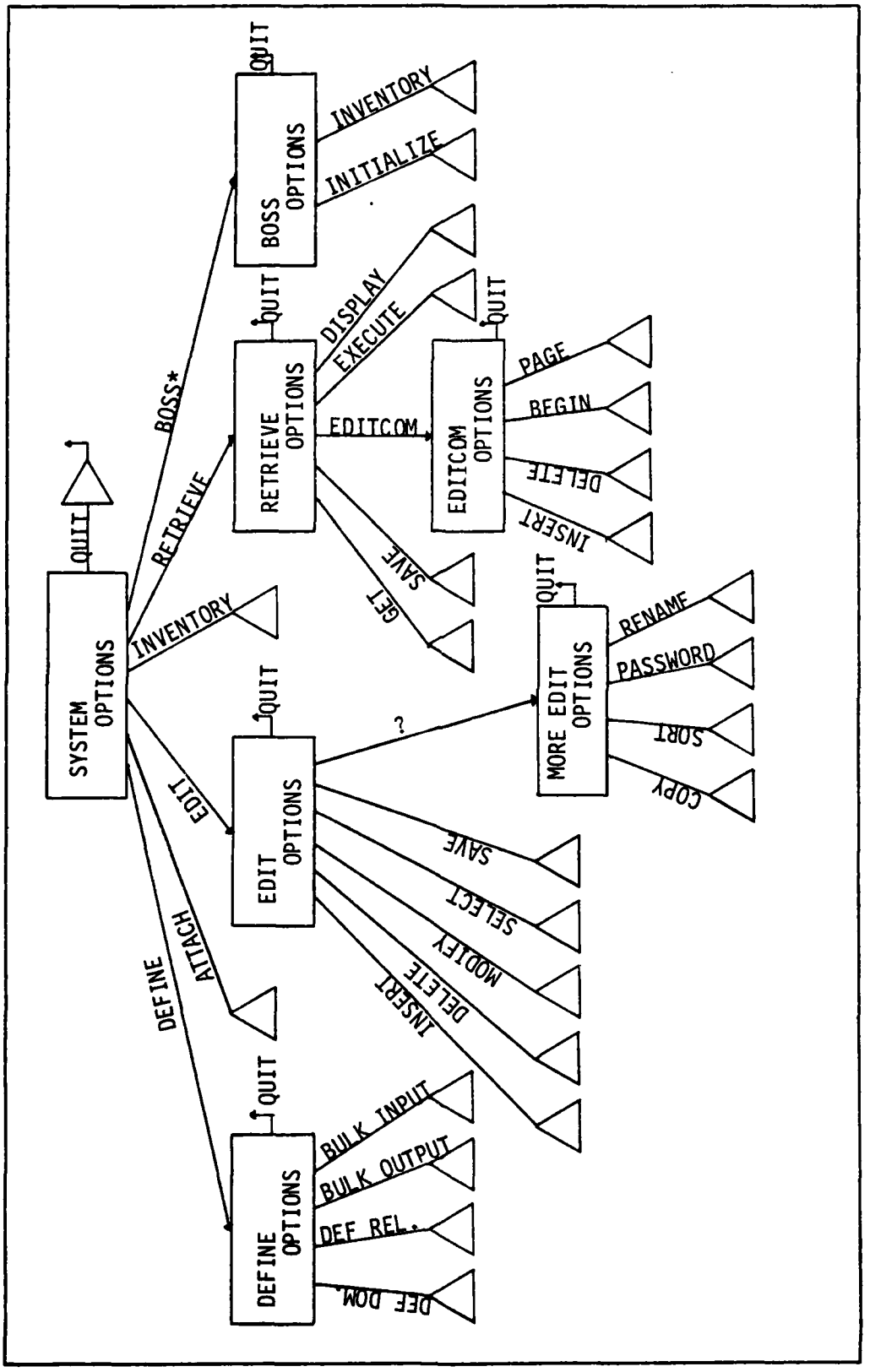


Figure B-1. Multi-level User Command System

the CONSOLE device. For example, the prompt line for the outermost level of the system is:

```
Sys Options:   D(efine), E(dit), R(etrieve), I(nventory),
               A(ttach), Q(uit), B(oss)
>
```

By typing "R" the user will "descend" a level within the structure diagram into a level called "RETRIEVE". Upon entering RETRIEVE, another prompt-line detailing the set of commands available at the RETRIEVE level of the system is displayed. The Q(uit) command causes the user to exit from RETRIEVE and "ascend" back to the outermost command level of the system. Now the user is back at the level in the system from which he started after initially executing the system. Some commands within the system prompt the user for more information, such as the name of a relation, a line number, etc. In these cases, the user enters the required information followed by a carriage return (<cr>). If an error is made in typing a portion of the information, the backspace key (or equivalent key depending upon the system configuration) may be used to "back over" and erase the erroneous part. If the user decides not to accept any information at all, "escape" from most commands is by entering zero characters; i.e., type <cr>. Unless otherwise stated, any input to a yes/no question besides a "Y" is considered a no.

Sometimes there are more commands available than would be reasonable to display at one time. When this is true, a question mark (?) will appear at the end of the line. Typing "?" will cause a different prompt to appear, such that more of the available commands will be displayed to the user.

1.2 Outermost Level Commands: An Overview

A. D(efine)

Typing "D" while at the outermost command level of the system causes the DEFINE segment to be brought into memory from disk. The user may, while in DEFINE, define domains and relations, and perform formatted input/output operations between mass storage and relations. See section 2 for details.

B. E(dit)

"E" places the user in a level of the system called EDIT. This section of the system contains commands used primarily for maintenance of relations. See section 3 for more details.

C. R(etrieve)

This command allows the user to formulate and execute relational queries on the database relations, and display the results. A workfile is used to hold queries, and commands include "getting", "saving", "editing", and "executing" this workfile. See section 4 for more details.

D. I(nventory)

Typing "I" at the SYSTEM level will cause a list of the domains which have been defined and a list of the relations which have been defined and attached (see E below) to be displayed on the CONSOLE.

E. A(ttach)

Type "A" at the SYSTEM level to enter ATTACH. The user is then prompted for the relation to attach. Currently only an attach flag is set in the relation definition. ATTACH will, in the next version, also request security passwords from the user if he does not own the relation.

(Refer to section 2 for explanation.) Relations must be attached before anything can be done with them.

F. B(oss)

This module is avaialable only to the database administrator (DBM). It provides special commands such as initialization. See section 5 for more details.

1.3 Starting the System

The system is started by executing the code file DATABASE.CODE. The disk which contains this file must remain on-line during the execution of the system, in order to permit segment swapping.

Upon execution of the system, a welcome message is displayed and the user is asked to enter an identification name. This name is associated with all relations created by the user and is maintained as part of the security system. The user is then asked if his system has 1 or 2 disk drives. If 2 drives is indicated, then the domain and relation definitions are assumed to exist on the disk in Drive 1, on a file called SETUP.DATA, else Drive Ø is assumed to contain the disk with SETUP.DATA. If all goes well, the system prompt-line is displayed; otherwise an error message is generated.

The database manager identifies himself with a special identification name. The DBM can create a file called SETUP.DATA if one does not exist.

1.4 Key Words and Names

Names in the system; e.g., relation names, attribute names, etc., may be any combination of 1 to 8Ø non-blank characters. Although not guaranteed to be harmful, user defined names should not be any of the following key words:

ALL	DIVIDE	JOIN	PRODUCT	UNION
AUG	FROM	MAX	PROJECT	VETO
BY	GIVING	MIN	SELECT	WHERE
COUNT	IN	ONE	SORT	$@$
DIFFERENCE	INTERSECT	OVER	SUM	

Type "D" at the SYSTEM level to enter DEFINE and the following prompt is displayed:

```
(1)  DEFINE DOMAINS
(2)  DEFINE A NEW RELATION
(3)  INPUT FROM MASS STORAGE
(4)  OUTPUT TO MASS STORAGE
(5)  QUIT

SELECT 1 - 5 --->
```

The individual DEFINE commands are invoked by typing the number to the left of the parenthesis. For example, "1" would invoke the DEFINE DOMAINS command.

2.1 1) DEFINE DOMAINS

Defines domains to be used in defining relations. The user is prompted to enter the domain name, the type of domain; i.e., Character, Integer, or Real, and the number of characters or digits to be allowed. The domain name must be unique. If other than digits are entered when required, then Ø is assumed; and if the maximum integer size of the machine is exceeded then the maximum integer is used. All yes/no questions must be answered with a "Y" or "N".

2.2 2) DEFINE A NEW RELATION

Define a new relation to consist of the following parts, each prompted for individually:

Relation Name -- must be unique

Attribute name/domain name pairs -- attribute names must be unique within the relation, domain names must have been previously defined. The user is also asked if a sorted directory is to be maintained for each attribute.

Primary key(s) -- the number of keys is entered, and then each attribute to be a key is entered. The attribute must have been defined and if a duplicate is entered, the user is allowed to prematurely quit specifying keys. This exists to allow the user some way to quit when he has entered a number larger than the total number of attributes.

Security passwords -- the ID password is automatically set to the identification name and the user is allowed to enter security passwords for:

READ -- prevents display of or relational operations on the relation.

DELETE -- prevents deletion of any or all tuples.

MODIFY -- prevents modification of any tuple.

INSERT -- prevents insertion of any new tuples.

Constraints -- the user is allowed to specify constraints on the attributes of this relation. These are used to further restrict the domain of an attribute. The constraint can be of the form:

$$\text{attribute} \left\{ \begin{matrix} > \\ < \\ = \end{matrix} \right\} \text{value}$$

or

attribute = (value1, value2, value3, . . ., valueN).

Multiple constraints can exist on an attribute.

2.3 3) INPUT FROM MASS STORAGE

This command has not yet been implemented. Its purpose is to read in data from disk file into a relation, specifying the format of the data. Data which is incompatible with the domain type or constraints of the relation attributes should be flagged as an error, and non-unique key values should also be flagged.

2.4 4) OUTPUT TO MASS STORAGE

This command has not yet been implemented. Its purpose is to write data in a relation to disk or to a printer in a particular format.

Type "E" at the SYSTEM level to enter EDIT and the following line is displayed:

```
Edit Options:  I(nsert), D(elete), M(odify), S(ave), R(esort), Q(uit),?
>
```

Typing "?" in response to this prompt displays more EDIT commands:

```
More Edit Options:  C(opy), S(elect), P(assword), R(ename), Q(uit)
>
```

The individual EDIT commands are invoked by typing the letter found to the left of the parenthesis. All EDIT commands have yet to be implemented; but a description of each is given here.

3.1 I(nsert)

Insert a tuple into a relation. The key value(s) must be unique in the relation and values must be in the domain and satisfy constraints of each attribute. The relation must have been attached and the INSERT password (if any) specified if the relation is not owned by the user.

3.2 D(elete)

Delete tuples from a relation. A single tuple may be deleted by specifying its key value(s) or a set of tuples if values are specified for one or more other attributes. When doing multiple deletes a VETO option exists to allow the user to individually decide on the deletion of each tuple. The relation must have been attached and the DELETE password (if any) specified if the relation is not owned by the user.

3.3 M(odify)

Modify tuples of a relation. The tuples to be modified are specified by giving values for any or all of the attributes. Then the values for the attributes to change are specified, a blank value indicating that the old value should remain. A VETO option similar to that in the D(elete) command also exists for M(odify). The relation must have been attached and the MODIFY password (if any) specified if the relation is not owned by the user.

3.4 S(ave)

Save newly created relations; i.e., those created as a result of relational operations on other relations (see section 4). The user can specify security passwords and integrity constraints on the relation, with tuples violating the constraints being deleted with user approval.

3.5 R(esort)

Sort the tuples of a relation. The relation must have been attached and the READ password (if any) specified if the relation is not owned by the user.

3.6 C(opy)

Copy a relation into another with the ability to change attribute names and the domains they are defined on. The copy will specify the mapping between attributes. If the domain type of an attribute in the receiving relation is incompatible with the data values of the attribute it is receiving then an error is reported and the COPY aborted. The only transformation of type allowed is integer to real. Key attribute(s) in the receiving relation must receive unique values. The relation being

copied must be attached and the READ password (if any) specified if the relation is not owned by the user, and the receiving relation must have been defined.

3.7 S(elect)

Select a tuple from a relation satisfying a specific constraint or perform a function on the relation. The tuple selected can be:

-- any tuple
-- one in which an attribute has either a maximum or minimum value.

The functions include:

--COUNT(attribute1, . . ., attributeN) which will give a numerical count of the unique occurrences of the attributes listed.
--SUM(attribute) will give the sum of the value in each tuple of attribute.
--AUG(attribute) will give SUM(attribute) divided by the total number of tuples in the relation.
SUM and AUG must have integer or real arguments.

The relation must be attached and the READ password (if any) specified if the relation is not owned by the user.

3.8 P(assword)

Change the security passwords of a relation attached and owned by the user.

3.9 R(ename)

Change the relation name and/or attribute names of a relation attached and owned by the user.

Type "R" at the system level to enter RETRIEVE and the following prompt line is displayed:

```
Retrieve ops:  G(et), S(ave), E(dit), X(ecute), D(isplay), Q(uit)
>
```

A concept central to the operation of RETRIEVE is the command file. A command file contains one or more relational queries. The command file can be created, modified, stored on disk, retrieved from disk, and executed. The commands which can reside in a command file are described below. Several examples are provided after that.

A. Union of two relations:

UNION relation1, relation2 GIVING relation3

where the first two relations must be union-compatible; that is, they have the same number of attributes and the ith attribute of one relation must be drawn from the same domain as the ith attribute of the other relation. Relation3 will acquire the attribute names of relation1. Relations 1 and 2 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation3 must be unique.

B. Intersection of two relations:

INTERSECT relation1, relation2 GIVING relation2

where all restrictions under UNION apply.

C. Difference or relative complement of two relations:

DIFFERENCE relation1, relation2 GIVING relation3

where relation3 = relation1 - relation2. All restrictions under UNION apply.

D. Cartesian product of two relations:

PRODUCT relation1, relation2 GIVING relation3

where attribute names in relation3 will be the same as those in relation 1 and 2 except that duplicate names will be prefixed by the name of the relation it came from. Relations 1 and 2 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation3 must be unique.

E. Join of two relations:

JOIN relation1, relation2 WHERE attr1 op attr2 GIVING relation3

where attr1 is in relation1 and attr2 is in relation2, op is =, <, >. The JOIN operation is a subset of the cartesian product where the condition of membership is specified in the WHERE clause. All restrictions under PRODUCT apply.

F. Project a relation over a subset of its attributes:

PROJECT relation1 OVER attr1,attr2, . . .,attrN GIVING relation2

where attributes not specified in the OVER clause will be eliminated and any duplicate tuples will be eliminated. Relation1 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation2 must be unique.

G. Select a subset of tuples from a relation:

SELECT ALL FROM relation1 WHERE condition GIVING relation2

where condition is a boolean predicate on the attributes of relation1 of the form a1 AND/OR a2 AND/OR a3 . . ., where each aN is of the form attribute op value, where op is =, <, or > . The expression may be fully parenthesized to indicate the proper precedence of the operators, but if not then AND has precedence over OR. One or more blanks or commas must be between each part of the command except that the left parenthesis may be flush against an item to its right, and the right parenthesis may be flush against an item to its left. Relation1 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation2 must be unique.

H. Divide a binary relation by a unary relation:

DIVIDE relation1 BY relation2 OVER attr1 GIVING relation3

where relation1 is a binary relation, relation2 is a unary relation, and attr1 is an attribute of relation1 defined on the same domain as the attribute in relation2. Relation3 will be a unary relation with attribute from relation1 not attr1. Relation 1 and 2 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation3 must be unique.

Examples: The following relations are used as the basis for examples:

Relation: part, Key: part#

part#	name	location	color
56	wheel	Miami	silver
78	cam	Boise	red
79	tire	Boise	black
100	seat	Dayton	green
711	fender	Miami	black
899	clock	Enon	red
1245	light	Boise	silver

Relation: shipment, Key:(part#,supply#)

part#	supply#	quantity
78	4567	890
100	45	900
100	546	50
100	4567	435
899	2309	1000
899	4567	13

Relation: shippart, Key: part#

part#	name	location	color
78	cam	Boise	red
100	seat	Dayton	green
899	clock	Enon	red
1000	cap	Dayton	blue

Using these relations examples of the output of the above commands is shown beneath the command.

UNION shippart, part GIVING upart

upart

part#	name	location	color
56	wheel	Miami	silver
78	cam	Boise	red
79	tire	Boise	black
100	seat	Dayton	green
711	fender	Miami	black
899	clock	Enon	red

(cont'd on next page)

part#	name	location	color
1000	cap	Dayton	blue
1245	light	Boise	silver

INTERSECT shippart, part GIVING ipart

ipart

part#	name	location	color
78	cam	Boise	red
100	seat	Dayton	green
899	clock	Enon	red

DIFFERENCE shippart, part GIVING dpart

dpart

part#	name	location	color
1000	cap	Dayton	blue

PRODUCT shipment, dpart GIVING ppart

ppart

shipment-part#	supply#	quantity	dpart-part#	name	location	color
78	4567	890	1000	cap	Dayton	blue
100	45	900	1000	cap	Dayton	blue
100	546	50	1000	cap	Dayton	blue
100	4567	435	1000	cap	Dayton	blue
899	2309	1000	1000	cap	Dayton	blue
899	4567	13	1000	cap	Dayton	blue

JOIN part, shipment WHERE part# = part# GIVING shipment-description

shipment-description

part-part#	name	location	color	shipment-part#	supply#	quantity
78	cam	Boise	red	78	4567	890
100	seat	Dayton	green	100	45	900
100	seat	Dayton	green	100	546	50
100	seat	Dayton	green	100	4567	435
899	clock	Enon	red	899	2309	1000
899	clock	Enon	red	899	4567	13

PROJECT shipment-description OVER color, supply# GIVING c-and-s

c-and-s

color	supply#
green	45
green	546
green	4567
red	4567

SELECT ALL FROM part WHERE location = Miami GIVING parts-in-Miami

parts-in-Miami

part#	name	location	color
56	wheel	Miami	silver
711	fender	Miami	black

SELECT ALL FROM shipment WHERE (supply# > 4000 AND quantity < 800)

GIVING s-q

s-q

part#	supply#	quantity
100	4567	435
899	4567	13

```
PROJECT shipment OVER part#, supply# GIVING ps#

ps#

part#     supply#
  78        4567
 100          45
 100         546
 100        4567
 899        2309
 899        4567

PROJECT s-q OVER supply# GIVING s#

s#

supply#
4567

DIVIDE ps# BY s# OVER supply# GIVING p#

p#

part#
  78
 100
 899
```

Each RETRIEVE command may be split between two or more lines in the command file if the split is made at a key word. For example, some of the ways the last command in the examples above could be split is as follows:

```
DIVIDE ps#      DIVIDE ps# BY s#      DIVIDE ps# BY s# OVER supply#
by s#           over supply#          GIVING p#
OVER supply#    GIVING p#
GIVING p#
```

The commands may be combined in any sequence to formulate one or more queries. For example, the query "Find the colors of all parts supplied by any supplier in quantity > 500" can be expressed as:

```
SELECT ALL FROM shipment WHERE quantity > 500 GIVING T1
JOIN part, T1 WHERE part# = part# GIVING T2
PROJECT T2 OVER color GIVING answer
```

The query is created and executed within a command file via the commands available at the RETRIEVE level.

4.1 G(et)

Get a command file from disk into the workfile. A workfile is simply a command file in memory. If the current workfile is not empty then the user must decide whether or not to throwaway the current workfile before getting another.

4.2 S(ave)

Save the workfile as a command file on disk. If a previous command file was obtained using G(et) the user is asked if he wishes to save the workfile with the same name. If a file already exists on the disk with the same name, the user must decide whether or not to destroy the file on disk before saving the workfile. If there is no room on the disk an error message is generated.

4.3 E(dit)

Typing "E" at the RETRIEVE level causes the following prompt-line to be displayed:

```
Edit ops:  I(nsert), D(elete), B(egin), P(age), Q(uit) --->
```

Edit is used to create and modify command files while they are in the workfile. After execution of each Edit command except P(age) the first 20 or fewer lines of the command file is displayed on the screen, preceded by a line number.

4.3.1 I(nsert)

Insert one or more lines into the workfile. If a workfile exists then the user is prompted for a line number after which the new lines should be entered. An entry of Ø indicates before the first line and a line number greater than the last line number indicates after the last line. If a line in the file is specified, that line is displayed at the top of the screen and the user allowed to insert lines until only a return is entered. The workfile is renumbered after the insertion.

4.3.2 D(elete)

Delete a line from the workfile. The workfile is renumbered after the deletion.

4.3.3 B(egin)

Display the first 20 or fewer lines of the workfile.

4.3.4 P(age)

Display 20 or fewer lines starting at a particular line number of the workfile.

4.4 X(ecute)

Execute the command file in the current workfile. If syntax errors are found in the file, they are reported to the user. This causes execution to be aborted; however, the user can indicate that the syntax of the rest of the file is to be checked for syntax errors while ignoring the command in which the error occurred.

Each query of the command file is executed and the result is put into the temporary relation specified by the user in the query. If the query is determined to do nothing, such as the union of a relation with itself, then the query is not executed and this fact is reported to the user.

Currently only the low-level procedure calls necessary to perform each query are output.

4.5 D(isplay)

Display the contents of a relation on the screen.

When quitting the RETRIEVE level, the user must decide whether or not to throwaway the current workfile, if one exists.

If the user has logged on with the special DBMID as his identification name, then access is allowed to the BOSS procedures as well as special priviledges throughout the system. The following commands are currently available in BOSS:

"E" -- exit BOSS.

"I" -- Inventory, same as that at the SYSTEM level.

"Z" -- Initialization of the system, which currently deletes all domain and relation definitions in memory. SETUP.DATA will also be initialized if the user quits the system in this configuration.

When logging on with the DBMID, all relations are automatically attached and presumed owned by the user.

CHANGING THE SYSTEM * SECTION 7

Please refer to sections 3.3.1 and 3.3.2 of the UCSD Pascal Version II.0 reference manual and current program listings before trying to change the system.

The program is currently divided into a main body segment, four segment procedures and a unit. Each segment is contained in a dummy program in order to permit separate compilation. The unit contains all types, variables, and procedures which are global to more than one segment. Thus, by including the unit in each dummy program, access is allowed to those elements. The format for each segment procedure is:

```
PROGRAM dummy-name;
USES COMMON;  (*the unit is named COMMON*)
SEGMENT PROCEDURE name(parameter-list);
  Local types, variables, and procedures;
BEGIN
  body of name;
END; (*name*)
BEGIN
END.  (*dummy name*)
```

Since the segments are separately compiled, the parameter list of the segment procedure, must contain all global variables accessed or modified by the procedure. The program or segment procedure which calls for each segment procedure must have a dummy segment procedure with the same name, so that it may compile properly. The format for the main body segment is:

```
PROGRAM main;
USES COMMON;
local labels, types, constants, and variables;
SEGMENT PROCEDURE name1( ---);
BEGIN
END; (*name1*)
SEGMENT PROCEDURE name2(---);
BEGIN
END; (*name2*)
.
.
.
```

```
other local procedures;
BEGIN
  body of main;
END.  (*main*)
```

The format for segment procedures which use other segments is the same as the previous format for a segment procedure except dummy segment procedures are included as local procedures.

Each segment procedure has a particular segment number from 11 to 15 associated with it. The main body segment has number 1 and the unit has number 1Ø. Other numbers are for Pascal use only. The way numbers are assigned to the segment procedures is in first compiled, first numbered order. Thus, in the above format for the main body segment name1 would be assigned 11, name2 assigned 12, etc. Therefore, in each dummy program used to define a segment procedure an appropriate number of dummy segments must exist before the defined segment to ensure that the segment count is the same. Thus, for example, the format for segment name2 would be:

```
PROGRAM dummy name;
USES COMMON;
SEGMENT PROCEDURE dummy name1;
BEGIN
END; (*dummy name1*)
SEGMENT PROCEDURE name2( --- );
  local labels, types, etc.
BEGIN
  body of name2
END; (*name2*)
BEGIN
END.  (*dummy name*)
```

The unit -- COMMON -- is compiled and placed in the system library using the librarian program, LIBRARY.CODE. (See section 4.2 of the UCSD Pascal manual.) When each program is compiled, the unit is retrieved from the system library and used in the program; however, each program must still be linked with the system library in order to bind the external

variable and procedure references into the unit. After each program is compiled and linked, then the librarian may be used to put all the segments together into one code file. Each code file containing the segment is retrieved and linked into the proper space using its assigned segment number into the overall file.

If a particular segment, including the main body segment, is to be changed then the steps to be followed are:

1) Change the source code for the segment.

2) Compile the program containing the segment.

3) Link the code file to the system library.

4) Using the librarian create a new overall file passing all unchanged segments to the new file and linking in the changed segment.

If the unit has to be changed then after compiling it and placing it into the system library, each program must be recompiled, linked to the system library, and then put together with the librarian.

APPENDIX C

BASIC PROCEDURES FOR IMPLEMENTING

CODD's RELATIONAL ALGEGRA (REF 12)

These procedures are designed on the following assumptions:

1. The permanent relations stored in the database (i) may or may not be stored in sort order on one domain, and (ii) may have no directories at all, or may have several directories over different domains.

2. Directories are small in size compared to relations.

3. The number of secondary storage page accesses can generally be reduced by analysis of directories to determine those pages containing tuples satisfying a given condition.

4. Temporary relations produced during expression evaluation may be either stored in their entirety or piped via a small buffer.

5. The number of page accesses will be minimized by always processing a stored relation in sequential order and making maximum use of each page while it is in main memory.

6. The execution of each procedure should be extravagant in neither time nor space.

The basic procedures are as follows.

1. PROJECT1 (Relation, domainset)
(Assumption: domainset contains the primary key domains of Relation)

Tuples from Relation are processed in the order of supply and only the domains on domainset are preserved in each output tuple.

2. PROJECT2 (Relation,domainset)
(Assumption: Relation will be supplied sorted on some domain in domainset.)

Tuples from Relation are processed in the order of supply and only the domains in domainset are preserved in each output tuple. All tuples having a given value on the sort domain are checked for duplicates before output tuples are piped upward.

3. PROJECT3 (Relation, domainset, sortdomain)

Tuples from Relation are processed to simultaneously remove domains not in domainset and to sort the resulting tuples on sortdomain; in the sorting process duplicates are removed. Output is piped upward.

4. SELECT (Relation, condition)
(Assumption: condition is a boolean expression of restriction and selection predicates on the domains of Relation).

We will say a subexpression is "resolvable" if pointers to all and only tuples satisfying the subexpression can be determined by directory analysis. Ths "resolution" of a subexpression is such a set of pointers.

The resolutions of all resolvable subexpressions in condition are determined. If the whole expression is resolvable, its resolution is sorted. Tuples referenced by the resolution are accessed sequentially and piped upward. Otherwise, by directory analysis alone, a "minimal" set S is constructed which contains pointers to at least all tuples satisfying condition. S is sorted. Each tuple referenced by S is checked for satisfaction of condition. This can be done by using known subexpression resolutions, direct checks of tuple values or both. Tuples satisfying condition are piped upward. More details for implementing these algorithms may be found in (1).

5. JOIN1 (Relation1, domain1, =, domain2, Relation2, Unary Relation)
(Assumption: Either Relation1 or Relation2 is unary. Unary Relation states which of the two relations is unary.)

For purposes of exposition we assume that Unary Relation is Relation1 - the other case is symmetrical. If Relation2 does not have a directory (say D) over domain2 then one is created. For each value in Relation1, D is searched to find pointers to tuples in Relation2 which have the same domain2 value. These pointers are stored in a set P. P is sorted into address order. Tuples referenced by P are accessed sequentially. Each tuple is concatenated with another copy of its domain2 value and piped upward.

6. JOIN2 (Relation1, domain1, = , domain2, Relation2)
If Relation1 and Relation2 are not already sorted on domain1 and domain2, then they are sorted appropriately. Relation1 and Relation2 are accessed sequentially looking for tuple pairs satisfying the condition domain1 in Relation1 = domain 2 in Relation2). Access is advanced along Relation1 or Relation2 depending on a comparison of domain1 and domain2. Tuple pairs satisfying the condition are concatenated and piped upward.

7. JOIN3 (Relation1, domain1, condition, domain2, Relation2, Sort Relation)
(Assumption: Sort Relation is either Relation1 or Relation2. This parameter specifies which input relation is to be stored in sort order on its "joining" domain. The other input relation will be piped. Condition is <, $\underline{<}$ $\underline{>}$, or>.)

For exposition purposes let us assume that Sort Relation is Relation1 - the discussion for the other case follows from symmetry. If Relation1 is not already sorted on domain1 then it is sorted accordingly. If a directory (say D) is not available for Relation1 over domain1, then one is created. Relation2 is input and the value of each tuple over domain2 is determined. The directory D is consulted to locate the beginning/end point of tuples which satisfy the condition.

(domain1 in Relation1 condition domain2 in Relation2)

Since Relation1 is sorted on domain1, these tuples can be accessed sequentially. Each tuple satisfying the above criterion is concatenated to the tuple from Relation2 and piped upward.

8. UNION1 (Relation1, Relation2)
(Assumption: Relation1 and Relation2 are both sorted over a common domain.)

Relation1 and Relation2 are merged together and duplicates are removed. Output is piped upward.

9. UNION2 (Relation1, Relation2)
(Assumption: Relation1 and Relation2 both have directories over a common domain.)

Select a domain over which both relations have a directory. From these directories obtain a list of pointers (P) to tuples which are potentially in the intersection of Relation1 and Relation2. Pipe upward tuples in one relation (say Relation1), copying all tuples which are in the potential intersection into a relation I. Now pipe upward tuples in Relation2 pulling out those that are in P and testing them for membership in I. Those tuples already present in I are discarded.

10. UNION3 (Relation1, Relation2)
(Assumption: Relation1 and Relation2 have a common domain, such that Relation1 has a directory over this domain and Relation2 is sorted over this domain, or vice versa.)

Select a domain D satisfying the above assumption. Assume Relation1 has the directory over D. Pipe upward tuples in Relation2 copying out into temporary relation I those tuples whose D values occur in the directory for Relation1, and maintain a list of pointers (P) into the directory to those values which occur in Relation2. Now pipe upward tuples in Relation1 filtering out tuples pointed to by P and which occur in I.

11. UNION4 (Relation1, Relation2)

If neither Relation1 or Relation2 have a directory then one is created. The procedure is then similar to UNION3 except a binary search on the directory is necessary to create I and P.

12. INTER1 (Relation1, Relation2)
(Assumption: As in UNION1.)

The relations are merged and only common tuples are piped upward.

13. INTER2 (Relation1, Relation2, Order Relation)
(Assumption: As in UNION2. Order Relation is either Relation1 or Relation2, and determines whether the output is ordered as Relation1 or as Relation2.)

The general operation is similar to UNION2. If Order Relation is Relation2, then the first pass is made through Relation1 to create I and P.

The next pass is through Relation2 and only those tuples occurring in I are piped upward.

14. INTER3 (Relation1, Relation2, Order Relation)
(Assumption: As in UNION3. Order Relation is as described in INTER2.)

The general operation is similar to UNION3. The first pass is always made through the relation without a directory over the common domain. Order Relation determines whether, on the pass through the relation with a directory, output occurs directly when common tuples are found or subsequently by passing up common tuples in I.

15. INTER4 (Relation1, Relation2, Order Relation)

The general operation is similar to UNION4. Order Relation determines the output order as in INTER3.

16. DIFF1 (Relation1, Relation2)
(Assumption: As in UNION1.)

Relation 1 and Relation2 are merged and only those tuples in Relation1 not found in Relation2 are piped upward.

17. DIFF2 (Relation1, Relation2)
(Assumption: As in UNION2.)

The general operation is as in UNION2. The first pass is always made through Relation2 to create I. Tuples are only output on the second pass when Relation1 is checked against I via P.

18. DIFF3 (Relation1, Relation2)
(Assumption: As in UNION3.)

The general operation is as in UNION3. If Relation1 has the directory then tuples are output only on the second pass when Relation1 is run off against I via P. However, if Relation2 has the directory, then tuples which are not in the potential intersection are output on the pass through Relation1. When Relation2 has been run off against I, additional tuples which are not in the real intersection are output.

19. DIFF4 (Relation1, Relation2)
(Assumption: As in UNION4.)

The general operation is similar to UNION4. The discussion in DIFF3 concerning output applies here also.

20. CARPROD (Relation1, Relation2, Order Relation)
(Assumption: Order Relation is either Relation1 or Relation2 and determines whether the output has the same order as Relation1 or Relation2.)

If Order Relation is Relation1, then Relation2 is stored. For each tuple in Relation1 a complete pass of Relation2 is made; each tuple is concatenated to the Relation1 tuple and the result is piped upward.

If Order Relation is Relation2, then the roles of Relation1 and Relation2 are reversed.

21. DIVIDE (Relation1, D.Set1, D.Set2, Relation2, SortD)
(Assumption: If Relation1 will be supplied sorted on some domain in the complement of D.Set1 then SortD = "SO" (stored order). Otherwise, SortD will be some domain in the complement of D.Set1, and this domain will determine the output sort order.)

Project the D.Set2 domains from Relation2 and store the Relation1 on SortD. Scan Relation1 sequentially and examine each block of tuples containing the same value in the sort domain. If all tuples in T occur in the D.Set1 domains of a block associated with the same tuple t in the non-D.Set1 domains, then output t. Find and output all instances of t in the block. Repeat for all blocks in Relation1.

VITA

Mark Roth was born August 12, 1957 at Elgin, Illinois. He attended Fenton High School at Bensenville, Illinois, graduating June 1975. He then accepted a four-year ROTC scholarship to attend Illinois Institute of Technology, graduating a year early in May 1978, with a B.S. in Computer Science, and a commission in the USAF.

He was immediately selected to attend the Air Force Institute of Technology. On December 15, 1979, he graduated with an M.S. in Computer Science.

He is a member of IEEE Computer Society and the ACM.

Permanent Address: 4N650 Church Road
Bensenville, IL 60106

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/79-14	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* THE DESIGN AND IMPLEMENTATION OF A PEDAGOGICAL RELATIONAL DATABASE SYSTEM		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mark A. Roth 2LT USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 15 DEC 1979 13. NUMBER OF PAGES 151
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

J.P. Hipps, Major, USAF
Director of Public Affairs

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Relational Databases
Computers and Education
Query language
Query optimization

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A relational database system was designed with the goal of obtaining as near optimal behavior from the system as possible. In addition, the database was to be implemented as a general purpose system but with specific provisions for teaching database management and manipulation. Toward these goals, investigations were made into previous studies in the literature. The advantages and disadvantages of relational systems wre explored, and based on certain criteria a relational algebra was chosen as the basis for the data manipulation and

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

definition language. Solutions to the problems of relational databases, including integrity, redundancy, and efficiency, were presented in this context. With this background, a top-down structured design of the system was completed. Techniques used to manage data entry, i.e., the user interface, and techniques to transform those inputs in order to optimize their execution were deveoped and implemented. These transformations formed the basis of an automatic programmer used to analyze and efficiently refine high level query specifications supplied by the user. This approach sought to minimize query response time and space utilization by: (1) performing global query optimization, and (2) coordinating sort orders in temporary relations.